UNIVERSIDADE DO PORTO • FACULDADE DE ENGENHARIA

Documenting Software With Adaptive Software Artifacts

Filipe Alexandre Pais de Figueiredo Correia

January 2015

Scientific Supervision by

Ademar Aguiar, Assistant Professor
Departmento de Engenharia Informática

In partial fulfillment of requirements for the degree of
Doctor of Philosophy in Informatics Engineering
by the ProDEI Doctoral Programme

Contact Information:

Filipe Figueiredo Correia
Faculdade de Engenharia da Universidade do Porto
Departamento de Engenharia Informática

Rua Dr. Roberto Frias, s/n
4200-465 Porto
Portugal

Tel.: +351 22 508 1400
Fax.: +351 22 508 1440
Email: filipe.correia@fe.up.pt
URL: http://invisivel.net

This thesis was typeset on an Apple® MacBook® Pro running Mac OS® 10.6 using the free LaTeX typesetting system, originally developed by Leslie Lamport based on TeX created by Donald Knuth. The body text is set in Palatino, a large typeface family that began as an old style serif typeface designed by Hermann Zapf and that was initially released in 1948 by the Linotype foundry. Other fonts include Sans, Latin Modern and Typewriter from the Computer Modern family, and Courier, a monospaced font originally designed by Howard Kettler at IBM and later redrawn by Adrian Frutiger. The LaTeX style was based on the one created by Hugo Ferreira for his own PhD thesis. Most charts were drawn using the `matplotlib` python library.

This work was partially funded by the FCT grant number SFRH/BDE/33883/2009, with the support of the European Social Fund (POPH) and of PARADIGMAXIS, S.A..

ISBN 978-972-752-168-5

Filipe Figueiredo Correia
"Documenting Software With Adaptive Software Artifacts"

. . . to my parents and brothers

This page was intentionally left blank.

Abstract

The label *knowledge work* applies perfectly to the craft of software development, since it focuses on acquiring, transforming and capturing knowledge in various forms – as different kinds of *software artifacts*. While the creation of source code is a key part of software developers' end-goals, knowledge is usually first captured in forms that are not as objective, not as structured, and not as close to a working software system. To an extent, this happens due to the nature of knowledge itself; ideas often have to be iteratively discussed, reasoned upon and written informally before they can become more structured kinds of software artifact.

Many of these artifacts may be classified as *documentation*, as they don't play a part in the functional aspects of the final, working, software system, but rather provide information about it, to assist its development, use, or maintenance. When referring to documentation we often don't refer only to free-text documents but to a diversity of artifacts, such as user stories, tasks, models and source code, among others.

Many approaches to software documentation exist and address different needs, with wikis, literate programming and code annotations as some of the most influential ones. They imply different trade-offs and allow us to identify a set of factors that should be taken into account when designing new documentation approaches. These factors include the choice of supporting mediums (e.g., file-based or Web-based), whether to use a single-source or a multiple-source approach, to which extent the artifacts should enforce a specific structure, and how can multiple artifacts be integrated and related.

An issue that is especially important in the context of this work is the evolution of the knowledge captured within these software artifacts. Although knowledge will freely evolve in the minds of a software development team, artifacts may not be as easy to adapt accordingly, especially when changes to their structure are needed. The types of software artifacts that imply a specific structure have been conceived bearing in mind a particular type of knowledge. They are likely very expressive in that domain, but not easy to change beyond their predefined structure. Free-text documents are particularly worthy of note for their capability of conveying information in a wide

range of domains but they hardly enforce any domain structure and usually imply a certain ambiguity and verbosity. On the other end of the spectrum, source code in a general purpose programming language is unambiguous and terse, but it can only convey information on the domain of computations. A lot of software artifacts lie somewhere between these two extremes.

Although developers create software systems to manage information (which are often denoted as *information systems*), they don't usually see the many products of their work as information that they have to manage, and their tools and environment as the means to do so. This research recognizes that to make the most of the information captured during software development, it must be regarded as a primary source of knowledge. Furthermore, this work draws inspiration from architectures used in the context of adaptive software to address three specific concerns – the expression of information structure, the maintenance of contents' consistency and the classification of those contents so that they can be found more easily. *Adaptive Software Artifacts* is an approach conceived to tackle these concerns by combining benefits of free-form contents with those of structured contents. It has the goal of reducing the barrier to define new types of structure (i.e., new types of adaptive software artifacts) and allows adaptive software artifacts to be derived from textual contents and combined with them as needed. It tries to make it easy for developers to structure contents that otherwise would remain free-form and, therefore, less useful.

The contribution of this work is fourfold: a) a patterns catalog that formalizes good practices and design considerations surrounding software documentation, information classification, flexible modeling tools and adaptive object-models; b) the Adaptive Software Artifacts approach to software documentation; c) a reference architecture and implementation that was used to verify the practicality of the approach and to help validate it; and d) a statistical experiment that can be replicated independently, with the goal of validating the approach.

The experiment was run once with students and the results revealed some benefits in knowledge acquisition when using documentation following the Adaptive Software Artifacts approach.

Resumo

O desenvolvimento de software é um *trabalho baseado em conhecimento*, já que tem como foco a aquisição, transformação e captura de conhecimento em várias formas – como diferentes tipos de *artefatos de software*. Embora o código fonte seja essencial nos objetivos dos programadores, é normal o conhecimento começar por ser capturado segundo formas que não são tão objetivas, tão estruturadas, nem tão próximas de um sistema de software funcional. Isso acontece pela natureza do próprio conhecimento; as ideias muitas vezes tem de ser iterativamente discutidas, pensadas e escritas informalmente, antes de se tornarem artefatos de software mais estruturados.

Muitos destes artefatos podem ser classificados como *documentação*, já que não têm um papel no funcionamento do sistema de software, fornecendo apenas informação sobre ele, que assiste o seu desenvolvimento, utilização ou manutenção. Ao nos referirmos a documentação, muitas vezes não nos referimos só a documentos de texto livre, mas a uma diversidade de artefatos, como *user stories*, tarefas, modelos e código fonte, entre outros.

Existem muitas abordagens à documentação de software, cada uma preenchendo necessidades diferentes, sendo os wikis, a programação literária e a anotação de código algumas das mais influentes. Cada uma apresenta diferentes benefícios e deficiências, e permitem identificar um conjunto de fatores a ter em conta no desenho de novas abordagens de documentação. Estes fatores incluem a escolha dos meios de suporte (e.g., ficheiros ou Web), se é usada uma abordagem fonte-única (*single-source*) ou fonte-múltipla (*multiple-source*), em que medida os artefatos devem forçar uma estrutura específica, e como podem múltiplos artefatos ser integrados e relacionados.

Uma questão de especial importância no contexto deste trabalho é a evolução do conhecimento capturado nestes artefatos. Embora o conhecimento possa evoluir livremente nas mentes de uma equipa de desenvolvimento de software, os artefatos podem não ser tão fáceis de adaptar de acordo, especialmente quando são necessárias alterações à sua estrutura. Os tipos de artefatos de software que implicam uma estrutura específica foram concebidos tendo em conta um tipo de conhecimento em

particular. São provavelmente muitos expressivos nesse domínio, mas não são fáceis de alterar além da sua estrutura pré-definida. Os documentos de texto livre são particularmente dignos de nota pela sua capacidade de transmitir informação num grande leque de domínios mas dificilmente garantem qualquer estrutura de domínio e normalmente implicam uma certa ambiguidade e verbosidade. No outro extremo do expectro, as instruções de código fonte conseguem ser bastante inequívocas e concisas, mas só conseguem transmitir informação no domínio das computações. Muitos dos artefatos de software encontram-se algures entre estes dois casos extremos.

Embora os programadores criem frequentemente software para gerir informação (frequentemente chamados *sistemas de informação*), normalmente não vêm os produtos do seu trabalho como informação que têm de gerir, e as suas ferramentas e ambiente como meios para o fazer. Esta investigação reconhece que para tirar o máximo partido da informação capturada no desenvolvimento de software, esta tem de ser considerada uma fonte primária de conhecimento. Este trabalho inspira-se em arquiteturas usadas por sistemas adaptativos para abordar três questões específicas – a expressão de estruturas de informação, a manutenção da consistência dos conteúdos e a classificação desses conteúdos para que possam ser encontrados mais facilmente. A abordagem *Artefatos de Software Adaptativos* foi concebida para endereçar estes desafios, combinando os benefícios associados a conteúdos livres de estrutura com os benefícios dos conteúdos estruturados. Tem como objetivo reduzir a barreira à criação de novos tipos de estrutura (i.e., novos tipos de artefatos de software adaptativos) e permite que os artefatos de software adaptativos sejam derivados diretamente de conteúdos textuais e combinados com eles conforme necessário. Tenta tornar mais fácil estrurar conteúdos que, de outra forma, se manteriam em texto livre e, portanto, menos úteis.

A contribuição deste trabalho tem quatro partes: a) um catálogo de padrões que formaliza boas práticas e considerações de desenho em torno da documentação de software, classificação de informação, ferramentas de modelação flexíveis e modelos-de-objetos adaptativos; b) a abordagem de documentação de software designada como Artefatos de Software Adaptativos; c) uma arquitetura e implementação de referência, usada para verificar a viabilidade da abordagem e para ajudar a validá-la e d) uma experiência estatística que pode ser replicada de forma independente, com o objetivos de validar a abordagem.

O desenho experimental foi corrido uma vez com estudantes e os resultados revelaram alguns benefícios na aquisição de conhecimento ao usar documentação com base na abordagem de Artefatos de Software Adaptativos.

Contents

List of Figures

List of Tables

Preface

A culpa não, não é do sol, se o meu corpo se queimar
A culpa não, não é da praia, se o meu corpo se ferir
A culpa é da vontade, que vive dentro de mim,
e só morre com a idade, com a idade do meu fim...
A culpa é da vontade...

António Variações

I am told that this preface should explain you, dear reader, how and why this work came to be. The truth is I can probably trace it to my early years as a boy, and my general interest in computer games and in something called the *ZX Spectrum 48K*, followed by some interest a few years later in another little box called the *Nintendo Entertainment System*. It amazes me the influence that these toys had on a whole generation, and I am sure this thesis is one more tiny ripple of the impact they had on the pond of my childhood.

But there were more important factors conspiring to bring me here. I'm sure that watching my father design, build or fix whatever was needed around the home had no small role on my interest for the idea of *engineering*. It seems obvious, in retrospect, but I remember being quite undecided by the end of high-school as to which area I would like to pursue in college. Ironically, the results of a psychometric test suggested that I should enroll in a humanities course, but I knew that that couldn't possibly work for me. If anything, that test made me more certain that I should turn towards engineering or science – I have always been one to tinker, to build, and to share the results with others.

Fast forwarding to a few years later, I am happy with the course of my professional life in the industry – therefore, what could have possibly led me to pursue a PhD? The good memories I had from university a few years before played a tiny part but the main motivations were first and foremost my love for software engineering and design and the challenge to overcome myself. Research meant a *carte blanche* to seek

new knowledge, free from the everyday self-*pressure* to deliver working, immediately useful, software. I take from this journey much more than what I have learned within my specific topics of research, from design patterns, to agile software development, to the bonds that I have created with others.

As to the specific topic of my research, it's difficult to look back and get a clear image of the path that first led me to it. Someone that I much admire asked me why have I chosen this topic and, to my own surprise, I replied that I didn't think it had really been me who found it – rather, it found me! That answer came spontaneously and stuck with me for the next few days, as it made more and more sense the more I thought about it. The topic of this thesis emerged over time, from the confluence of several seemingly unrelated subjects that I am fond of. It starts with the early contact in my professional life with the notion of *literate programming* by the hand of my employer and friend Alexandre Sousa, who had himself contact with the concept during his PhD. It is influenced by my efforts at ParadigmaXis to have my team all collaborate through the same platform (a software forge), which included the creation of documentation for users and for developers using a wiki. My advisor Ademar Aguiar would rekindle my interest in these topics again years after. I was also influenced through my contact with topics from the domain of information science, in the context of my most enduring project at ParadigmaXis, aimed at cataloging and retrieving archive documents and their metadata. Finally, this work also stemmed from the research of my colleague and friend Hugo Ferreira on the adaptive object-model architectural pattern, with whom I've collaborated often during the time he was pursuing his own PhD.

I have received the help and encouragement of many people along this journey and my appreciation is beyond words. My deepest gratitude has to go to my advisor, mentor and friend, Ademar Aguiar, who was always able to make me refocus on what was most important when I found myself off the right track – thank you for your understanding and for believing in me even when I felt lost. In the category of mentors and friends I also have to deeply thank Alexandre Sousa, who was the responsible for my first job in the industry and supported me to pursue a PhD while working for ParadigmaXis – I know that not many would have given me such an opportunity, thank you!

I also have to thank Hugo Ferreira and Nuno Flores, for their friendship and companionship – this work would not be the same without your support and our joint brainstormings sessions. My thanks goes also to many other co-workers who, along the years, helped me grow professionally many times beyond myself – Alexandre Pinto, José Vilaça, Hugo Silva, Fátima Pires, João Ferreira, Tiago Cunha, Aurélio Pires,

Bruno Matos, Pedro Abreu, José Porto, Diogo Lapa and Ricardo Almeida, among others.

A special thanks has also to go to my teachers, colleagues and university co-workers: to João Correia Lopes, who encouraged me to write my first academic paper right after finishing my licentiate in 2002; to Eugénio Oliveira, Augusto Sousa, Raul Vidal, João Pascoal Faria, Ana Paiva and Rui Maranhão, for their friendship and for making me feel welcome back at FEUP after my years in the industry; to Tiago Boldt and Fábio Pinho for their friendship and support; to Idalina Silva and Marisa Silva, for their welcoming smile and readiness to solve all sorts of issues. I also have to thank João Araújo from *Universidade Nova de Lisboa* and (again) to João Pascoal Faria for accepting to participate in this PhD's steering committee and for their valuable feedback.

During this time I have participated of several communities. I have to give a special thanks to the patterns community, for their insights and for making me feel so welcome. My thanks go out to Joseph Yoder, Rebecca Wirfs-Brock, Eduardo Guerra, Lise Hvatum, Bob Hanmer, Linda Rising, Christian Kohls, Christian Köppe, Richard Gabriel, Peter Sommerlad, Ralph Johnson and Brian Foote, among many other friends and conference buddies.

To my longtime friends, Carlos, Ana, André and Eduardo, thank you for not giving up on asking if *"is it ready yet?"*.

My family, even those that remain only in my memory, were always with me for inspiration, support and encouragement, and must also have a place in these paragraphs – to my mother Filomena, my father Jorge, my brothers Nuno and João, my uncle and aunt Paulo and Judite, and my grandparents Jorge and Judith. To Branca for her valuable help proofreading this dissertation.

Lastly, I thank FCT – Fundação para a Ciência e a Tecnologia – and ParadigmaXis S.A., for funding this work.

Filipe Figueiredo Correia

Porto, Portugal, July 2014

Chapter 1

Introduction

Knowledge is the main *raw material* of software development. In a software project, knowledge is captured as multiple kinds of artifacts and, very often, as free-text documents. These enable to capture practically any knowledge that may be important to communicate or preserve for the future, because they allow recording information independently from any hard structural constraints. This makes text documents a very flexible kind of software artifact. Notwithstanding, free-text documents have their drawbacks – their narrative nature isn't necessarily the most helpful when readers are quickly trying to reconstruct the mental-model that authors tried to capture; maintaining text contents consistent can be very expensive as it requires continuous review; and finding information about a specific topic in a body of text contents may be difficult. These liabilities are all symptoms of the absence of a domain structure, which is also what enables free-text documents to be so flexible.

The work presented in this thesis addresses specifically the improvement of software documentation but spans topics such as knowledge capture, collaborative systems, software evolution and software development environments. It looks into techniques used to build software documentation, their benefits and drawbacks, and defines an approach to document software.

The present chapter starts by briefly establishing the context and scope of this doctoral work (Section 1.1) and goes on to explain its research goals, contributions and experimental findings (Sections 1.2 and 1.3). After presenting the work itself, it explains how this document is organized, and what readers may expect from the chapters that follow (Section 1.4).

1.1 Software Documentation

Software developers' knowledge is in constant change throughout a project's lifetime, and is recorded as software artifacts of different kinds. Knowledge on a given subject usually starts rather fuzzy and may be captured in free-form (e.g., as a textual document). As more knowledge is gained, and some notions become more clear, developers tend to capture knowledge using artifacts with a richer structure, like tasks on a project planning/tracking tool, source code, etc.

However, such a transition is not peaceful. One of the reasons is that it's usually expensive to *repackage* information – after capturing it as one type of software artifact, changing to a different kind of artifact usually implies a manual translation – and some information might even not have a clear place in the new form, possibly implying information loss. Sometimes, foreseeing that the change to a more structured kind of artifact will be needed, developers try to create such an artifact from the very beginning, but they may be committing prematurely to a specific information structure that will not be useful later on. Worse still, it may require some information to be made-up just to satisfy the tool in use, which may easily become *noise* in the future. Essentially, this sums up the choices faced by software developers: a) to capture contents free of domain structure (e.g., free-text documents) or b) to capture contents as specialized artifacts, with a predefined domain structure (e.g., source code, tasks, etc.). The former allows great flexibility and evolvability of the contents at the expense of precision and terseness, which makes contents more expensive to find and maintain consistent, while the later makes contents more precise and terse, but they usually demand a predefined form and are difficult to change beyond that form.

Software artifacts are, inevitably, always one step behind the perception of the team of how the software should actually be. The work described in this thesis focuses on the information that supports a software development endeavor and on how to easily mold it to support new needs.

One of the values of the agile manifesto is *"people and interactions over processes and tools"* [BBvB+01] – we believe this stems from the fact that too often developers are lead and locked into a workflow and specific information forms by the tools they work with. The key goal of this work is to empower developers to easily mold the information they work with, or rather, to easily adapt it to reflect their knowledge and the way it is used in the project.

1.2 Research Goals and Contributions

The goal of this work is to support the act of documenting a software system, by a project team, using two classes of contents: those that are free of a domain structure, such as textual documents, and those whose domain structure can be captured but may likely need to evolve.

Free-text documents are already extensively used to document software system, but developers need a way to structure the contents and share the result without having to commit to a software artifact with a fixed structure. With this goal in mind, an approach was defined to combine the benefits of textual contents with those of specialized software artifacts: it allows to express *domain structure*, like most software artifacts do, but also allows that domain structure to be adapted (i.e., evolved) to new forms if and when the need arises. This approach was named *Adaptive Software Artifacts* and is presented in detail in Chapter 4.

The results of this work arise from the thesis that:

> *Capturing software knowledge with the Adaptive Software Artifacts approach makes information easier to be consumed, created and evolved, especially in the context of medium-to-large projects.*

This thesis was decomposed into more specific research issues that are detailed in Chapter 4 and resulted in the four main contributions that follow.

A Patterns Catalog. Multiple patterns were documented throughout the research to formalize good practices and designs on software documentation, information classification, flexible modeling tools and adaptive object-models.

An approach to software documentation. The Adaptive Software Artifacts approach embodies some of the good practices and designs that were documented as patterns, to address the key concerns of this research.

A reference architecture and implementation. The Trac software forge [Edga] was extended through a plugin to support the Adaptive Software Artifacts approach. It aims to prove the concept of the approach and can be used as reference by other tool developers.

A statistical experimental designed to validate the approach. A statistical experiment was designed to validate the effects of the approach experimentally and with the goal of being easy to replicate. This experimental design was already used in an academic setting to gather initial results.

1.3 Experimental Findings

The statistical experiment was designed with a specific focus on knowledge acquisition issues. A group of MSc students participated of a run of the experiment that produced a set of promising results. These results showed some benefits of using documentation following the Adaptive Software Artifacts approach. Namely, they revealed that subjects spent significantly less time performing a sequence of software development tasks when given access to software documentation using the approach, when compared with subjects performing the same tasks with access to *regular* documentation. In particular, subjects spent less time on the platform searching and consuming contents. The documentation following the approach was found to be more concise and easier to understand and browse.

1.4 Thesis Overview

An outline of the chapters in this dissertation and their relative dimensions is depicted in Figure 1.1. The present chapter sets the overall context and objectives of this document and briefly introduces the general theme, motivation and results of this research. The next two chapters make a review on several related topics. The topic of software documentation is introduced first, in Chapter 2, establishing the importance of knowledge in software development and its relation with software artifacts, how they are designed, and the techniques and tools currently used to handle them. Chapter 3 follows with the topic of software evolution, and puts into perspective how software artifacts evolve and existing work on how software is made more adaptive.

The research problem and how it is approached will then be described by Chapter 4. This chapter enunciates the thesis statement, enumerates the specific research issues that underly the research problem, and establishes the set of research outcomes and the validation method of this work.

The following chapters describe the research outcomes in more detail. More

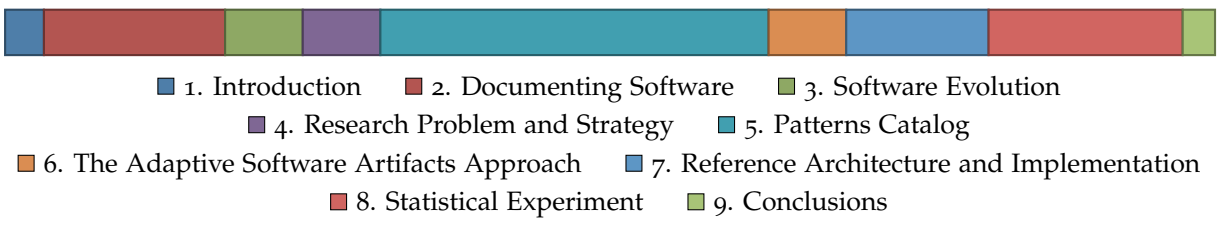


Figure 1.1: Overview of the dissertation's chapters and their relative dimensions.

specifically, they present a patterns catalog (Chapter 5), the Adaptive Software Artifacts approach (Chapter 6), a reference architecture and implementation (Chapter 7), and a statistical experiment to validate the approach (Chapter 8). The final chapter reviews all the contributions and explores some possible future directions and improvements to this work (Chapter 9).

This dissertation uses some typographical conventions to help its readability that deserve to be clarified. References to other works appear enclosed by [square brackets]. When expressions related to programming are used, such as class or method names, they are shown using a `mono-spaced` font. *Italics* are used to emphasize expressions of particular importance in their contexts and to highlight quotations and publication names. **Boldface** is used to highlight expressions that match concepts existing on a related figure or section, or to highlight the titles of paragraphs or enumerations. When patterns are mentioned, and very particularly in the patterns catalog (Chapter 5), these conventions are extended: Small Caps are used to refer to pattern names, *italics* are used to highlight the problem and solution statements, and **boldface** is used when referring to the names of the forces that shape the patterns' solutions.

Chapter 2

Documenting Software

An overview and summary of the main topics of this chapter may be found in Figure 2.1, which can be used as a chapter *roadmap*. The chapter starts with a review of the role of **knowledge** in software development projects, its relation with **information**, the importance of sharing, evolving and preserving knowledge over the project's lifetime, and what this means for **software artifacts** (Section 2.1). Next, it reviews approaches and tools used to handle software artifacts (i.e., **source code**, **models** and **free-text documents**) and **software documentation**, with the intent of identifying popular documentation solutions and to look in detail into the approaches that inspire

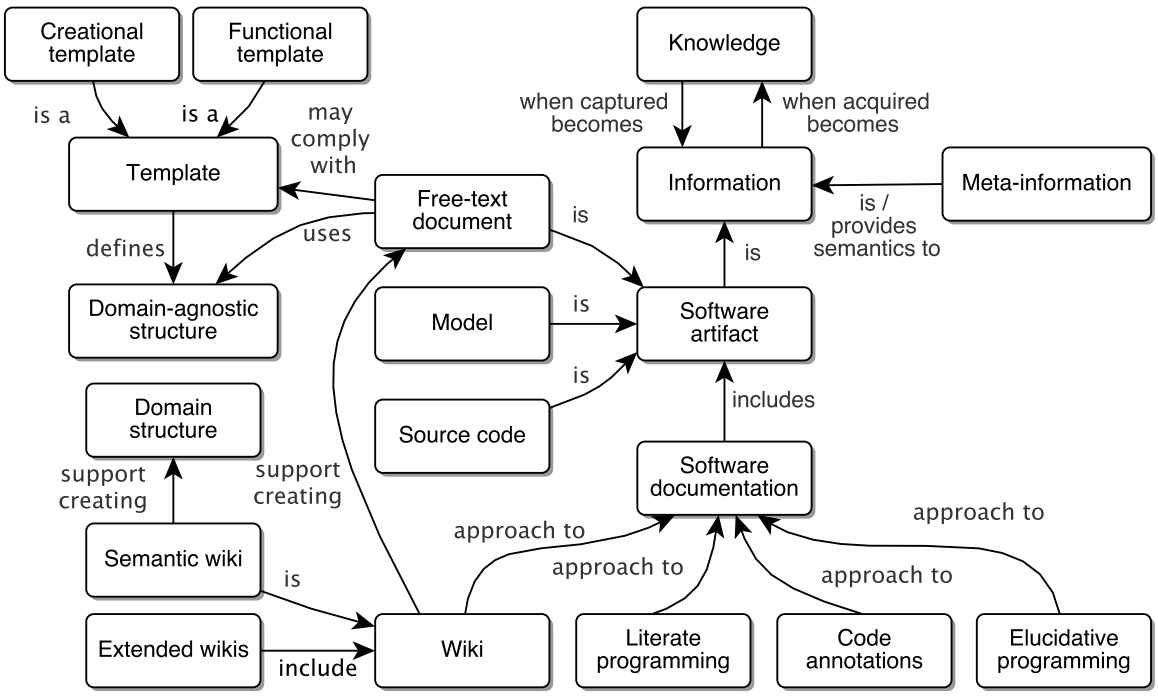


Figure 2.1: Concept map of software documentation topics.

this research (Sections 2.2 and 2.3). The remaining of the chapter is dedicated to concerns common to the design of software artifacts in general and the environments used to manage them, analyzing the several forces at stake (Sections 2.4 and 2.5).

2.1 Software Knowledge

The notions of knowledge capture and knowledge acquisition are not always consensual and have evolved over the years [Mul96]. For the purpose of this thesis, **knowledge capture**, or representation, is seen as the process of conveying knowledge in a medium, and, doing so, transforming and encoding it as **information**. On the other hand, **knowledge acquisition** is the process through which a human actor gains knowledge, that is, the process of learning and understanding information. This duality is depicted in Figure 2.2.

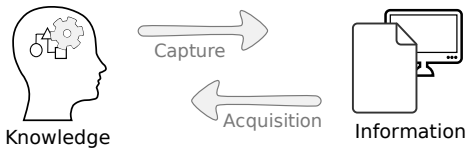


Figure 2.2: Knowledge capture and acquisition.

Research in the cognitive science domain has dealt with several knowledge-related issues. It has allowed to identify different types of knowledge and strategies through which they are acquired, with results useful on different domains of human activity, and in particular in that of software development [Rob99, RAMF04, GR13].

Software developers are knowledge-workers, as their activity revolves around creating, distributing and applying knowledge [Dav05]. In fact, software development can be said to be a knowledge-intensive activity in which there is as a progressive crystallization of knowledge with the very specific goal of ultimately obtaining instructions to be executed by a computer [Rob99]. The difficult part of this process is not necessarily the production of programming language statements *per se*, but often the discovery of knowledge that will allow the developers to produce them. This is essentially an activity of learning, or acquiring knowledge [Arm00], in which **software artifacts**, that are no less themselves a manifestation of knowledge, go through different levels of formality: from unstructured information, as textual documents and verbal communication, to source code statements that will be consumed by computers.

2.1.1 Knowledge Sharing and Preservation

Software developers are usually part of a team, which means that knowledge is worked collaboratively. Being able to share it effectively among team members is a key concern, as it enables to reduce overall knowledge acquisition efforts, and helps in the process of establishing a shared understanding that allows the team to work towards the same goals.

Knowledge sharing may happen by capturing it – that is, by recording it, and making it available to others – or it may happen through direct communication between peers.

By capturing knowledge, we are making it available to others and our future selves. The importance of this is that ideas, in the minds of team members, are not easy to maintain. There are a number of events that may disrupt keeping knowledge by memory alone: team members leave the project, new team members appear, existing team members simply forget knowledge that they once had, and they have to recover it. Regaining knowledge may involve repeating the process that has originally led to it, which frequently has a high cost.

Although this work addresses knowledge capture, it is worth noting that the value of direct communication should not be dismissed. Some development processes in fact advocate face-to-face conversation as essential to sharing knowledge on a short term effectively [BBvB+01, BA04]. As software projects tend to be increasingly distributed, software development tools and environments are also starting to include features for direct communication between peers [CHRP03, STvDC10].

2.1.2 From Knowledge to Software Artifacts

A great part of the effort in the development of software is in collecting information – from verbal conversations and written materials – which is then reasoned upon, transformed, and captured into concrete software artifacts. These can be of different types, such as **free-text documents**[1], **models** and **source code**, among others.

[1] The Cambridge Dictionary defines document as *"a paper or set of papers with written or printed information, especially of an official type"*, or *"a text that is written and stored on a computer"*. Even though this definition reflects a common understanding of what a document is, it is very broad and not especially useful to us. In practice, we may distinguish between *free-text* and *structured* documents. While the first refers to textual contents, made mainly by non-domain-specific elements such as sections, paragraphs, lists, figures, tables, etc., the second usually refers to domain-oriented formats, such as data-centric XML dialects. In the context of this work we shall refer to documents always as sets of primarily free-text contents, unless otherwise noted.

Two levels of captured knowledge may be considered: **information** and **meta-information**. While the information of an artifact addresses the particular subject that the artifact intends to represent, meta-information describes information itself, contextualizing it, and conferring it additional semantics. Simply put, meta-information is information that describes information and that can be used to define an information frame to which the information obeys or should obey. Or, in other words, it can be used to add structure to the information.

Different kinds of software artifacts provide different kinds and amounts of structure. While artifacts are usually bound to a certain degree of structure that cannot be changed, they frequently allow additional structural elements that are open to being authored. The process of capturing such structure is one of carefully organizing and classifying knowledge.

Take source code artifacts as an example – they need to obey to the syntax of the programming language that is being used, even if developers are free to define certain structural elements, such as classes in an object-oriented language. Considering a user manual as a second example, its authors likely obey the structure of free-text, that is, they are led to use elements such as sections, paragraphs, lists, figures, etc. Furthermore, if authors wish to create a specific document **template** (e.g., for requirements documents), a second level of structure is made available, establishing a concrete set of sections to be used for that kind of document.

Choosing a type of software artifact to capture a piece of knowledge always depends on the structure that it enables to express. For example, when creating source code, developers will try to express the solution in the programming languages that they are using, and one particular language may allow them to be more expressive in some parts of the solution. If it's a task that they are trying to capture, recording a new task in the project planning/tracking tool may be the best choice. On some occasions, though, there will be no clear choice of a specific kind of artifact that fits the knowledge to capture, and developers will try to record it in the most suitable one that they have available. A key factor in this choice is the proximity between the information (structural and otherwise) that software artifacts allow to express, and the mental models of the knowledge to be captured. The closer they are, the easier is to capture that knowledge.

The structure provided in the context of free-text documents is usually domain-agnostic, enforcing only a layout form. When using a template, authors are providing an additional frame that may come closer to the information's domain, but they may still be able to change the template to suit their needs. This shows how flexible free-text

documents can be, as they allow to express virtually any topic.

Despite this, free-text documents are intrinsically limited when trying to capture elaborate structures, and are sometimes combined with more specialized artifacts, in order to reach a better balance between flexibility and expressiveness [AD05b].

Expressiveness[2], the ability to convey the intended knowledge in the most complete and concise way, is a key factor to determine how effective knowledge capture is. It's not, however, the single criteria used by developers. As tools support the creation and use of artifacts, the choice for a specific type of artifact cannot be made without considering the maturity and functionality provided by the implied toolsets. Time (hence, cost) is also a variable to consider, and it may not be prudent to capture absolutely all the available knowledge. As a potentially time consuming activity, the effort and benefits of capturing knowledge should always be weighed [Bri03, Agu03].

2.1.3 Evolving Knowledge and Artifacts

As we will address further in the following sections, software systems can hardly be seen as immutable entities. The knowledge of project stakeholders, including that of developers, about a system under construction is always subject to change. This implies that the several artifacts captured at a given moment in time may have to be adapted to new understandings.

But, although knowledge may evolve, artifacts may not be easy to adapt accordingly, especially when structure changes are needed. The types of software artifacts that imply a specific structure have been conceived bearing in mind a specific type of knowledge. They are likely very expressive in that domain, but not easy to change beyond their predefined structure. The fact is that knowledge on a given subject frequently starts off as vague and only gradually becomes more concrete – only at a later time can the contents be structured effectively. If a given type of artifact is not capable of expressing additional structure, it may be impossible to update maintaining the same level of expressiveness of when it was first created.

[2] The meaning of *expressiveness* is difficult to pin down. The definition used in this work tries to approach the intuitive meaning of expressiveness, considering it as a combination of three factors: a) The ability (or inability) to convey information in a given domain; b) The degree of unambiguity (or ambiguity) of a conveyed information; c) The concision (or verbosity) of a conveyed information.

2.2 Software Artifacts

Software artifacts are both the products of software development and the *things* that developers work with. They may be themselves part of the final set of deliverables to be built; they may describe or support the process of developing software, and how it unfolds; and they are capable of describing the function and design of software, and therefore be used in the creation of other software artifacts.

A large variety of software artifacts may be considered, including project plans, requirements, design diagrams, models, source code, bug reports, user stories, graphical design resources, and translation resources, among many others. Some approaches to handle them are independent from the type, but usually different types of artifacts entail specific challenges and solutions. Free-text documents account for many of these different kinds of artifacts, as they usually allow their document structure to be shaped at will, and artifacts with different structures may easily be considered to be of different types.

Section 2.3 will go specifically into the topic of **software documentation**, which may comprise not only **free-text documents** but also other kinds of artifacts. This section goes into more details specifically about **source code** and **model** artifacts.

2.2.1 Source Code

The creation of abstractions has been repeatedly used in software development as a way to move the focus from the details of the hardware to the domain of the application being built. Traditionally, this abstraction process has assumed the form of higher-level development platforms and programming languages.

Source code is always the main focus of software development. Even though some approaches tend to focus on other kinds of artifacts (Literate Programming focuses on textual descriptions, Model-Driven Engineering focuses on models, etc), source code is needed to instruct machines what to do. In fact, some argue that source code is the only artifact that one can really depend on, as it *doesn't lie* – if inconsistent with other artifacts, source code is the artifact to go to for the actual program behavior.

Although source code can be extremely expressive in its own computation-oriented domain, the overall knowledge from which it was derived is very weakly recorded within its form. It exists only implicitly, rather than explicitly, being therefore difficult, if even possible, for the developers to reconstruct the original mental models at a later time from the source code alone [GAO95, Sta10].

2.2.2 Models

Models imply a higher level of abstraction than that provided by source code. A model is a simplified representation of a problem domain, capable of more accurately reflecting the developer's knowledge in that specific domain. This means they can normally capture knowledge more explicitly than source code, even though they don't represent every single detail of the actual problem to be solved.

The creation and use of models poses numerous challenges to software developers [SV06, TPT09]. We shall look into specific issues, such as *Consistency*, *Semantic Opacity* and *Abstraction Domain*.

Consistency

Models may be used to derive other artifacts at design-time, or they can be used by applications at run-time. Each approach has different merits, and should be chosen according to the constraints at hand. The first approach has traditionally been used more extensively, although it may easily be a source of concerns if the derived artifacts are not kept in sync with their original sources.

From a perspective focused on the quality of the structured contents, models are made out of information and meta-information that are expected to comply to each other. When this fails, a consistency issue may also be said to exist. These issues are discussed in a broader sense on Section 3.1.

Semantic Opacity

Using models implies making information obey to a semantically rich structure, but the use of modeling languages still frequently results in capturing knowledge as simple image diagrams, with the sole goal to be used for human consumption. Such artifacts can only be said to be the result of a modeling activity, and can't truly be said to be models. This is an important distinction, since tools that handle models and those that handle (model-derived) image resources may take considerably different approaches and provide different benefits.

Domain Expressiveness

General purpose modeling languages, such as UML, allow to model a substantial part of software systems. UML accomplishes this by supporting several types of models, targeted at different domains – specialized models may allow a significantly

more expressive representation of knowledge. For this reason, the need for a higher expressiveness in a given domain may easily spawn the need for different, specialized, models. Each possibly needing supporting tools with different requirements.

2.3 Approaches to Software Documentation

As mentioned earlier, free-text documents are a very flexible way of recording knowledge. They allow to express almost any type of information, can have different degrees of structure and be adapted to the specific needs at hand.

The primary goal of software documentation is to capture and share information about a software system in its different dimensions, including sometimes the activities that surround its making and use. We define documentation of a software system as any form of captured information about that system that may help its developers and users understand it. It serves as a communication medium between the members of a team and plays a key role in program comprehension [VN04]. Furthermore, when referring to documentation we are not merely referring to (textual) documents; documentation may comprise a very heterogeneous set of artifacts, and how they are organized and combined greatly depends on the type of documentation to produce.

2.3.1 Diversity of Software Documentation

Authors deal with several forces when taking the decision of what kind of software documentation to produce [BKM00, HHT01, AD07, AD11].

Different writers. Documentation is produced by different participants, with different roles in the software process. Programmers, architects, project managers, product managers, technical writers, and others, may all play a part in the production of documentation, as they are the most knowledgeable in their areas of expertise. Although depending on the way the entire development process is designed, the creation of documentation is usually a collaborative activity.

Different audiences. Documentation has different target audiences. The participants in the software process may be themselves the target audience of the produced documentation, as it can be others, external to the software development process. In either case, readers may possess distinct levels of knowledge, and documentation intended for them should take this fact into account.

Different subjects. Documentation may be used to describe any of the facets of software. It may in fact be used to describe other artifacts, further contextualizing

them or using them to support more elaborate descriptions. Source code, models, and the working product itself, are some of the types of artifacts that may be the subject of documentation, or be made part of the documentation themselves. Different levels of abstraction can be touched simultaneously.

Different notations. Different kinds of information are better communicated by using different representations. While for some kinds of information a textual description may be the most appropriate, other kinds of information may be better conveyed in other ways; for example, by using diagrams, or source code examples. Target audiences can also influence the choice of notation to include in a particular document, as they will understand it more easily if they are already familiar with its notation.

Different forms. According to the context at hand, by balancing the previously presented alternatives, one can conceive and structure a free-text document in a way that it's most effective. Some recurring types of document structures address typical documentation structuring needs: scenarios, design patterns and pattern languages, system overviews, user manuals, tutorials, contextual help, frequently asked questions, cookbooks, recipes, hooks and motifs, among others.

2.3.2 Wikis

Wikis are systems for collaboratively authoring Web pages, which makes them tools with a very wide scope of application. The first wiki was created by Ward Cunningham in 1995[3] [Cuna] and numerous other implementations have since spawned from the same set of founding ideas [LC01], which can be summarized as a set of design principles enunciated by Cunningham [Cunb, Cuno6]:

Open – Should a page be found to be incomplete or poorly organized, any reader can edit it as they see fit.

Incremental – Pages can cite other pages, including pages that have not been written yet.

Organic – The structure and text content of the site are open to editing and evolution.

Mundane – A small number of text conventions provide all necessary formatting.

Universal – The mechanisms of editing and organizing are the same as those of writing, so that any writer is automatically an editor and organizer.

Overt – The formatted and printed output will suggest the input required to reproduce it.

[3] The first wiki engine, named WikiWikiWeb, is nowadays still on-line at `http://c2.com/cgi/wiki`.

Unified – Page names will be drawn from a flat space so that no additional context is required to interpret them.

Precise – Pages will be titled with sufficient precision to avoid most name clashes, typically by forming noun phrases.

Tolerant – Interpretable (even if undesirable) behavior is preferred to error messages.

Observable – Activity within the site can be watched and reviewed by any other visitor to the site.

Convergent – Duplication can be discouraged or removed by finding and citing similar or related content.

Using wikis as tools in software development dates back to their origins, and due to the benefits they bring for collaborative authoring and knowledge management, they are, nowadays, very popular and massively used by developers. Wikis have been found to be especially suited in agile environments, and for the creation of lightweight software documentation [Rö3, Agu03].

Structuring Contents

The several artifacts used during software development are frequently related, even if those relations may not always be explicitly recorded. Free-text documents may themselves be software artifacts, but they can easily describe other artifacts and the existing relations between them, while combining them and providing a context that may be difficult to capture otherwise.

As suggested in Section 2.1.2, information authored on a traditional wiki follows a free-text document structure: it is modeled using sections, paragraphs, lists, figures, etc. The structure of a free-text software document usually starts of as very simple and only gradually tends to become richer. Computers may process free-text according to its units, namely, to present them in different ways, to give each of them a different degree of importance during a search operation, etc. Information may thus be handled to some extent apart from what it is actually about.

But often free-text is not enough, and authors may want to go beyond its constituent units, and structure information according to its domain.

Expressing and reusing information structures may take different approaches, as is further detailed in the sections that follow: **templates** make possible the reuse of free-text document structures; **extended wikis** allow to go beyond a free-text model and express information according to an additional set of fixed domain-oriented structures; and **semantic wikis** enable authors to express information according to

domain-oriented structures, and to some degree allow to extend such structures or create new ones.

Templates

Some wiki engines use the term *template* to refer to the overall visual layout of the wiki, but it is important to note that we will adhere to a different definition. We regard templates as a way of structuring contents, so that they may be reused across wiki pages that assume the same document form. Although the concrete approach may vary, this kind of template is supported by several wiki engines.

Using the classification introduced by Di Iorio et al [IVZ08], structure-based templates may use two different approaches:

> **Creational** – Sometimes referred to as seeding pages, these templates allow establishing the initial contents of a wiki page. They are used to add an initial structure to a page that is being created, which authors then fill in with further contents. Creational templates are used only at page creation time, and no connection is maintained beyond that moment, between the page and the template that gave origin to it, which means that updating the template at a later time will not produce any effect on the pages that derived from it. The same syntax is frequently used for both pages and templates, so the same mechanisms are used to author both. Wiki engines such as MoinMoin[4], Confluence[5] and Trac[6] provide support for creational templates.
>
> **Functional** – Functional templates define fragments of content that may be invoked from wiki pages. Their objective differs from creational templates in that they're not meant to provide structure for an entire page, but only to a part of it. When invoked, the contents of a functional template are transcluded to the page; authors supply parameters that will fill the gaps established by the template. Functional templates also differ from creational ones in that there is a persistent connection between the templates and the pages where they are used, and changing the template will automatically affect how its pages are rendered. Like creational templates, this kind of template often uses the same syntax as regular wiki pages, but only to some degree: they extend that syntax to support the invocation of templates and the specification of the parameters that it may

[4] Available from http://moinmo.in/
[5] Available from http://www.atlassian.com/software/confluence/
[6] Available from http://trac.edgewall.org/wiki/PageTemplates

receive. Functional templates don't have such a broad support as creational ones; a notable example of a wiki engine supporting functional templates is MediaWiki[7].

Di Iorio et al have also suggested an improved approach to creational templates [IVZ08, IVZ09]. It uses the same mechanism as creational templates while maintaining a strong connection between pages and templates, and allowing template changes to have an effect on the pages they're connected to. They call this approach *lightly constrained templates* – compliance between a page and the structure defined by its template (i.e., consistency) may be assessed at any time after the creation of the page.

Also worth mentioning is the DBpedia project[8], as an approach to leverage the structure provided by functional templates. DBpedia focuses on extracting structured information from Wikipedia and distributing it in such a way that it may be queried and linked [AL07]. It does so by inferring a domain-oriented structure from functional templates[9], as template parameters are usually meaningful within the information's domain.

Wikis Extended for Software Development

Some wiki engines have been developed specifically with software development in mind. They extend the typical free-text model of wikis and provide several richly-structured artifacts [ADP03, Atl, Edga, fit].

Extended wikis take into account the semantics of the software artifacts that they support, and allow to combine those different types of content (text, models, source code, etc), thus taking a broad view of the software development process [Agu03, AD05b].

Most of the wikis described in the following paragraphs go beyond the notion of a wiki page, to support different knowledge capture needs and software development activities.

XSDoc. XSDoc [ADP03] uses both source code and models in the context of the wiki environment. It uses a multiple-source approach, storing documentation and source code in different files, even though these contents may be presented close to each other on a wiki page. This separation makes it easier to combine

[7] Available from http://www.mediawiki.org/wiki/MediaWiki
[8] Accessible at http://dbpedia.org/About
[9] It uses Wikipedia's *infobox* functional templates.

and reuse different kinds of content, as well as the use of different tools and environments.

Trac. Both the linking and inlining of content can be seen as forms of reuse. Trac [Edga] (also a multiple-source approach), makes the inline reuse of documentation possible [Kan]. Unlike XSDoc it does not directly allow the inclusion of UML models, but provides integration of a wider scope of software artifacts, such as source code files, issue tracker items, project milestones, and source code revisions, among others. Trac is an example of a Software Forge[10] and, as such, goes much beyond the features of a wiki.

FitNesse. FitNesse [fit] is also an interesting wiki engine for software documentation. It is classified by its authors as a software development collaboration tool, a software testing tool, a wiki and a Web server, all in one. FitNesse allows the creation of automated acceptance tests in a collaborative way – inside wiki pages –, including stakeholders in the process of defining inputs and executing the tests. Wiki pages may include both textual descriptions and tests, which can be ran from within the wiki page context. Tests may be seen as software documentation themselves, or as one more kind of software development artifact that may be further described.

Galaxy wiki. Xiao et al. have integrated source code and textual descriptions on a wiki, in a way that both are editable from the wiki environment [XCY07]. Wiki pages correspond to classes, and can include documentation as textual descriptions that are, this way, bound to a specific class. In fact, textual descriptions and source code for a given class are internally stored in the same file, as regular commented source code. Despite the fact that this approach doesn't consider artifacts other than textual descriptions and source code, this concept can be easily expanded to include other kinds of artifacts.

Semantic Wikis and Variants

Templates support the creation of new contents and structure that are mostly based on the free-text model of wiki pages, and extended wikis incorporate other kinds of contents that fit pre-established domain-oriented structures. But these two models are not enough when authors need to create new types of domain-oriented structures

[10] Software Forges are discussed in detail in Section 2.5.

themselves. Semantic wikis try to answer this need, allowing authors to deal with free-text contents and, at the same time, allowing them to create their own domain-oriented information structures. The kind of structure, and the way in which it's used, varies among semantic wiki engines.

Wikis denoted as semantic often use Semantic Web technologies, such as the *Web Ontology Language*[11] (OWL) and the *Resource Description Framework*[12] (RDF), to define their own meta-models. Our use of this term is thus very broad, as some wiki engines are not consensually referred to as semantic but provide very similar approaches using meta-models that are different of these.

As a general rule, semantic wikis take a top-down approach to structure contents, forcing *types* to be defined before the contents themselves are created. These contents can be added later and typed, for example, through text annotations. Very few engines take a more flexible bottom-up approach and allow structuring the contents first and only deriving types afterwards. Semantic wikis also tend to associate types to pages, so one may say that the notion of a page is overloaded with that of an instance of a type.

Different semantic wikis strike a different balance between the advantages of having structured information, how expressive can authors be in information capture, and how easy it is to capture it. The following paragraphs present some concrete examples.

Semantic MediaWiki. Building on the foundations of MediaWiki, Semantic MediaWiki allows users to structure page contents by explicitly adding annotations, which are included on the page's contents [KVV06]. These annotations extend MediaWiki's model by allowing to turn text contents into attributes and to express roles on hyperlinks, turning them into relations. MediaWiki's categories are seen as classes supporting type inheritance (i.e., is-a relations), as categories relate to other categories hierarchically.

KiWi. Knowledge in a Wiki (KiWi) [SBD+08] supports the creation of structured information by allowing pages to be typed, according to a set of pre-established RDF types [W3C08]. To each page, authors may then explicitly add values according to the type's allowed properties and relations. Structured information on a page is thus bound to the specific type of that page. Property values are not annotations of the contents; although they are bound to a specific page, they're added separately from the page's textual contents.

[11] The full specification of OWL may be found at http://www.w3.org/TR/owl-features/.
[12] The full specifications of RDF may be found at http://www.w3.org/standards/techs/rdf.

XWiki. XWiki is also a semantic wiki: In XWiki a page may be bound to a class (i.e. a page type), which establishes the structure of this and other pages. Types and their underlying structure are created using the concept of *ClassSheets*, at a moment prior to the creation of any of its eventual page instances. The creation of pages is then bound to the structure defined, which is provided by a user through a form-based user-interface [DGB07].

Moki. Ghidini et all define a reference architecture for wiki-based conceptual modeling tools [GRS12]. They present Moki as a *Conceptual Modeling Wiki*; a wiki in which pages are the universal building block for model elements (entities and properties). It has the goal of combining both unstructured and structured contents, with the benefit of allowing to build models from a textual description and of documenting model elements with such descriptions. It uses OWL and BPMN[13] as the reference modeling languages, and tries to foster collaboration between domain experts and knowledge engineers.

Tricia. Tricia is described as a hybrid wiki [MNS11, BMNS11]. It acknowledges that the approach of most semantic wikis is to let inexperienced users enter textual content that are later structured by domain experts. Tricia distances itself from this approach, and includes only a subset of semantic wikis features. Rather than trying to use semantic web technologies to their full, it tries to provide features that can be easily used by both roles – knowledge engineers and domain experts. The overall goal is to support modeling, through attributes and type tags.

MikiWiki. MikiWiki [ZVB11] also has the notion of page types, which it uses to link structured contents to specific templates and layouts, and supports incrementally moving from informal text to structured contents. It doesn't use semantic web technologies to model the contents, but rather relies on JSON, XML or other text-based conventions that users may choose. It also introduces the concept of *mikinugget*, which are user-editable rendering strategies, for the different kinds of structured data. Although this concept may likely be applied to other areas, the key motivation for Miki is software development.

[13] The full specification of the *Business Process Model and Notation* (BPMN) can be found at `http://www.bpmn.org/`.

2.3.3 Literate Programming

As enunciated by its creator [Knu84], the ultimate objective of Literate Programming (LP) is to make computer programs comprehensible by human beings, accomplishing this by switching the focus that is traditionally given to source code artifacts to documentation artifacts. The fundamental idea behind LP is that, when writing programs, one should not instruct a computer what to do, but rather explain to human beings what the computer will do.

An alleged benefit is that programs written this way are works of literature, or works of art [Knu84]. This claim may be seen with suspicion by those concerned with the practicality of this technique but, in fact, the usefulness of LP goes considerably beyond the aesthetic and literary side of documentation [Ham94].

Literate programs are built by describing pieces of the program at the same time they are developed, and by connecting them as a web of related ideas. The result is a unified document, combining several fragments (*chunks*) of source code and documentation, disposed not as a set of assorted blocks of information, but following a line of reasoning. Contents are arranged in the order in which they are written, improving their ability to be understood [Knu83, KC02]. This can be a benefit in what concerns readability, when compared to organizing source code according to its own structure.

LP involves the use of several languages; being required at least the use of a document formatting language, a programming language, and a unification language that assists in combining the previous two.

The reader's attitude towards program understanding also has to be different; the most effective approach to understanding a literate program is not an exploratory one, but by reading the literate document, in a book form [Ham94].

Several tools following a LP approach have been developed since the concept was conceived, with varying levels of success. None, however, has reached the mass acceptance expected by some authors, given the alleged benefits [Wyk89, PKB04]. Some barriers to the adoption of LP have been the dependence of LP tools for specific programming and formatting languages [vAK92], and the lack of methodological integration in the software lice-cycle [CB91]. Some of the most noteworthy LP systems are `WEB` [Knu83], `CWEB` [KL02] and `noweb` [Ram94]. The usage of these systems is quite similar; they supply two core operations: `weave` and `tangle`. While the former is used for generating a human readable form of the documentation, the later is used to generate the source code, in the form that is accepted by the compiler. These two kinds

of generated artifacts are created from the same literate document, commonly called a *web document*[14]. The *weaving* and *tangling* processes are illustrated by Figure 2.3.

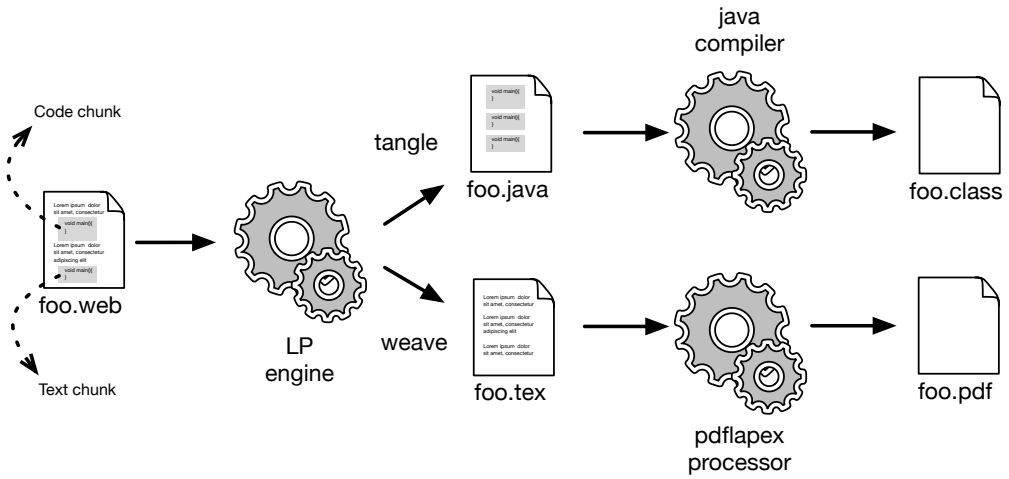


Figure 2.3: Literate programming operations.

Since its first appearance, research has shown Literate Programming to possess some additional virtues, as well as drawbacks. The following list presents the features of LP as been seen by its creator and according to subsequent research on this topic.

Verisimilitude. Documentation and source code are written and stored together, physically close to each other (i.e., as consecutive blocks in a file), and written at the same time [Wyk90]. This has the advantage of easing the production and maintenance of documentation, as the documentation is presented very closely, whenever the source code is modified. The resulting documentation may thus more easily be kept consistent.

Arrangement. Using a traditional approach, source code is organized in the way in which it is accepted by the compiler, according with the intent of the developer, and using the programming language's syntax. The order in which the code is kept usually isn't the same by which it was written, and neither can it be easily re-organized to better expose the reasoning that lead to its creation. LP allows (in fact, it requires) the re-organization of source code, so that it accompanies the line of thought of the documentation, following the psychological arrangement in which it may be better understood.

[14] No relation to the World Wide Web (WWW) exists, as the concept of Literate Programming predates the WWW in at least a decade. These web documents shouldn't be confused with the hypertext documents that form the WWW.

Despite the benefits, this feature also implies some important liabilities. The developer should only interact with the rearranged document, but this makes debugging more difficult. The compiler acts upon the *tangled* source code, and usually reports errors and warnings by referring to line numbers, but these will not match the line numbers of the web document that the developer has contact with. This happens because the developer and the compiler have different views of the code [KC02] leading to an effect commonly referred to as *referential opacity* [Thi86]. It frequently forces the developer to inspect the *tangled* source code files whenever he needs to match compiler-reported line numbers to a particular instruction, hindering his efficiency. This suggests a *leaking abstraction* [Spo02] yet to be resolved in traditional LP tools.

As with most powerful features, being able to create and rearrange source code and documentation fragments can be easily misused. One such case happens by abandoning the programming language's structuring mechanisms in favor of the chunking mechanism provided by LP. Chunks don't provide scope and supply very weak interfaces, hindering reusability and maintainability if used to replace language constructs. Using LP does not reduce the need to use the appropriate abstractions provided by the programming language [Ham94].

Another issue is that a single psychological arrangement may not be enough. Effective communication with different audiences may be better achieved using different arrangements.

Readability. By supporting and automating the creation of indexes, table of contents and cross-references, and by pretty-printing source code, LP tools allow a great readability of documentation on paper [Ham94]. On-screen, however, other approaches are needed to ensure a good readability. On-screen readability is important in several situations, and is fundamental during the development process [Kna96, Agu03].

Consistency. Software documentation is as valuable as much as it reflects reality, thus the importance of keeping it up to date, and consistent. As mentioned before, LP allows an easier maintenance of consistency, by keeping related pieces of knowledge physically close to each other – i.e., documentation for a given block of code is kept in the same file and in the same sequence as that block of code.

In spite of this benefit, consistency is still kept by visual inspection, and still requires such effort from developers, as other techniques do.

Contents integration. When using LP, documentation and source code are seen as a whole. This means they are produced at the same time, and are explicitly interrelated. As mentioned before, contents are organized sequentially, as a single document. In fact, several types of content, other than code and textual descriptions, can be integrated this way, such as figures [SC93], models [AEQ99], formal specifications [And01], etc. Several types of documentation can be produced following this approach, including external documentation, as user manuals [Thi86].

One should use the kind of artifact that is most appropriate to convey each piece of knowledge. A particular kind of artifact can be said to be more appropriate if it is more expressive than the alternatives (Section 2.1.2), or because it is simply of better use – e.g., if they are required as input for other software engineering tasks and tools.

Free-text has its place and is frequently used to capture what can't be expressed by other artifacts, to achieve a comprehensive documentation. An example is the use of free-text to overcome the trivialization of requirements in UML diagrams [AEQ99] – relative importance or priority of different requirements are not expressed in a UML diagram, but can be added by using textual descriptions.

Tools integration. Although being desirable that LP tools are both general and powerful, these objectives are not easy to reconcile [Thi86]. This poses a difficult problem, as two of the reasons that are most frequently pointed out for the lack of acceptance of LP are its lack of generality (e.g., its binding to a specific programming language) [vAK92] and its lack of integration with modern environments. These issues have both been addressed before (the first one with partial success [Ram94]), but are still major barriers to acceptance.

Appropriate tools support would allow to minor the debugging problem (see the *arrangement* issue discussed above), and would make possible to give LP environments the same capabilities of modern IDEs in what concerns code navigation, refactoring and on-screen readability, among others.

Quality. While it is as easy to write poor documentation using LP tools as is by using traditional tools, when using LP tools the developer is more aware of a reader, to whom the code is targeted. He will, thus, be more aware of the

importance of creating such documentation, and more inclined to writing it in a greater quantity and with a greater quality [Ham94].

Concerning code quality, it is also claimed that literate programs have fewer bugs, due to the extra attention that source code receives. By carefully explaining the objectives of a chunk of code, one can produce code of better quality, as errors will more easily come to one's attention [HS98].

Overhead. LP introduces a certain overhead. It may be made less noticeable with the help of the right tools and on small projects, but it's never negligible, especially for large or distributed projects [vAK92, Agu03].

The definition of what LP is has evolved over the years, and different people tend to emphasize different features when describing it [Knu83, Thi86, Ham94, Smi01, Agu03, PKB04]. A trend may be observed though: in its latest incarnations LP has moved its focus, from a way to produce software documentation as a work of literature, to a way of organizing, integrating, recombining, and maintaining the consistency of contents.

The evolution from the initial concept has spawn some variants, such as *Literate Modeling*, *Reverse Literate Programming* and *Theme-based Literate Programming*, and inspired the creation of other documentation techniques, such as *Code Annotations* and *Elucidative Programming*. Each of these approaches can be said to establish a different model for how to structure and organize software. They are detailed in the following sections.

Literate Modeling

The concept of Literate Modeling was introduced by Jim Arlow [AEQ99, AN04, Arl06] as a natural evolution in the use of LP. With models playing an increasingly important role in software projects there's the need to include them as first class artifacts. Model-driven development takes models as a key part to the creation of a working system, but they may also be used simply as a way to document an implementation [Tho06].

Although, at its first form, Literate Modeling was focused on UML models, it has introduced the idea that artifacts beyond code and textual descriptions may also be used with Literate Programming.

Reverse Literate Programming

One of the benefits of using LP is having documentation with a good readability on paper, but on-screen readability is not one of its strengths. On the screen, developers

do not read code in a sequential way, and rather do it selectively, like an encyclopedia, using source code structures and control flow to navigate to the intended information. Reverse Literate Programming [Kna96] has much in common with Literate Programming, but relies on the active text elements provided by an integrated development environment, like folding, linking and bookmarking. Active text elements are used to support several concepts of LP, like presented in Table 2.1, which was taken from [Kna96].

Active Text	**Literate Program**
Fold element	Section with a macro definition
Collapsed source of fold element	Source code part
Comment on fold element	Documentation part
Link element	Relation between sections

Table 2.1: Representation of active text elements as a literate program.

This approach provides a much better on-screen readability and the use of all the interactive features of an IDE, while still allowing to produce a printable (*web*-like) literate document. One other declared advantage is that the *tangle* operation is no longer necessary, as active text elements are encoded in the source code as special characters, which are ignored by the compiler. In a way, the concept of LP is reversed, as the literate document is assembled when needed, and the main artifact is the source code.

There is, however, a price to pay for a Reverse Literate Programming approach. Documentation and code may no longer be arranged in whatever order may be the best for program comprehension, being included in the printable literate document following the predefined sequence of source code.

Theme-based Literate Programming

The need for multiple arrangements of contents to co-exist was mentioned above, as a way to target different audiences. This is the main issue addressed by Theme-based Literate Programming (TBLP) [KC02]. TBLP allows the combination of different types of chunks (code segments, figures, textual descriptions, unit tests, etc) by relating them with different types of connections. This combination of chunks is made according to a *theme*, which is a way of sequentially organizing those chunks according to a specific psychological order. Themes are a way of providing multiple documentation views of a system.

Seeing all contents as *chunks* allows arranging documentation in any order, as it allows to do the same for source code, and makes it easier to decouple textual descriptions from the code fragments they describe. This also means, however, that verisimilitude isn't always achieved, with disadvantages towards consistency-keeping tasks. This liability may be minimized when using live and semantically rich relationships between artifacts [KC02], so that changes in one artifact highlight the need for updates on the ones that relate to it, allowing to keep the overall system documented and consistent.

2.3.4 Code Annotations

This technique was initially inspired in LP, as documentation is generated from a unified representation of textual descriptions and source code. However, it is also fundamentally different from LP, as textual descriptions exist in the form of source code comments. This means that the unified representation of textual descriptions and source code is itself valid and compilable source code, avoiding an additional *tangle* phase. When comparing to LP, it is also important to highlight that writing code annotations is not the same as writing free-text contents, as they depend on the structure of source code files. As such, it misses one of the main benefits of LP, which is the possibility of reordering documentation according to an intended psychological arrangement.

Code annotations are primarily used for creating API documentation, and don't address all the issues that LP tries to address. Having said this, it has shown to be quite successful in this niche, and has helped increasing the awareness on the need for documentation, and showing how it can enhance program comprehension.

The widespread use of this approach has been much the merit of Javadoc [Fri95], which is a tool supporting this functionality for the Java programming language. It is one of the first known uses of the technique, along with Autoduck [Artoo], a tool created in 1993, which supports code annotations in C++, and Doxygen [vH97], a tool initially released in 1997, and that now supports a wide range of languages, including Java, C#, Objective-C and Python, among others.

2.3.5 Elucidative Programming

Although this technique was inspired by LP, there are fundamental differences between the two. Elucidative Programming (EP) addresses the documentation issue from a more pragmatic standpoint; while LP satisfies the need of publishing programs as technical documents, EP tries to address the everyday maintenance needs and program comprehension [Nøo0]. The main goal of EP is to directly support software maintenance tasks, by providing documentation that can be used effectively from within the development environment. In fact most EP features may only be streamlined with such an integrated environment [Ves03].

Unlike LP, EP allows attaching explanations to a program without modifying the source code, thus not requiring the additional *tangle* and *weave* phases. Furthermore, EP does not demand the re-arrangement of source code; instead, it allows the creation of documentation following two different styles [VN02]:

> **Linked.** This style provides a mechanism for defining bi-directional relations between source code and documentation sections. These relations are presented as hyperlinks; while browsing the code, the developer can at any time explore the relations by asking for the related documentation, and vice-versa.
>
> In an EP approach, documentation and source code exist as separate entities and are connected by referencing syntactical elements of the programming language or through special code markers embedded in comments.
>
> **Inlined.** The ultimate goal of EP is not the creation of a printable document that sequentially explains the system. However, some kinds of documentation don't fit a linked model of reading, being more adequate a sequential presentation of contents (e.g., tutorials, cookbooks, etc). EP's inlined documentation was conceived to handle these kinds of documents, allowing to author textual descriptions and to combine them with code fragments, following a line of thought. Source code fragments are not merely copied to the documentation, but rather referenced, so that no inconsistencies may be introduced between virtually identical artifacts.
>
> When using an inlined style it is still possible to benefit from some linked-style features. Words used in textual descriptions can be turned into hyperlinks that reference specific code elements.

References between artifacts can potentially be used, not only for user navigation purposes, but also for error checking features that detect inconsistencies caused by the evolution of source code or other software development artifact.

Another alternative to handle inconsistencies was approached by an extension to EP that considers the possibility of different paces on the evolutions of source code and documentation, and embraces this reality as a common scenario to be dealt with, instead of prevented [VN05].

2.4 Designing Software Artifacts

As we have suggested in Sections 2.2 and 2.3, different types of software artifacts often imply different challenges and require different approaches. Yet, they do share some common concerns that shape their design and that should be taken into account when creating new solutions.

2.4.1 Supporting Medium

The most common medium used to support software artifacts is the filesystem. A filesystem allows handling different *things* (e.g., files), of different types (i.e., file formats), leaving to applications the way such things are actually encoded. A vast amount of tools has been designed around this assumption, with version-control systems and source code editors as some of the most common.

But other supporting mediums may be found beyond files. Some software artifacts have also been managed and supported by Web-based information systems, and this approach has become increasingly popular. Wiki engines are a good example of these systems; artifacts (e.g., wiki pages) are not identified by a file path but by a Web address, and the Web browser is the primary tool to handle them. Some of such Web-based environments integrate distinct tools for collaboratively developing software and are frequently referred to as Software Forges [REM+09]. They are described in detail in Section 2.5.

2.4.2 Single and Multiple Source Approaches

File-based contents can be classified as single-source or multiple-source, depending if several kinds of content (e.g., source code and textual descriptions) are encoded in a same file, or if each type of content is created as an independent artifact.

Traditional approaches keep documentation and source code as autonomous artifacts, in what concerns their writing, reading and storage. This may lead to consistency

problems, as the need to replicate content is likely to manifest (e.g., to copy source code fragments to documentation).

Single-source approaches [Smio1, Aguo3] join different kinds of content in the same file, and are effective ways of solving this problem. They do, however, bring other difficulties; particularly if joining distinct formats in the same file leads to new, mixed or non-standard formats that require specialized tools.

Multiple-source approaches [Smio1, Aguo3] keep different kinds of artifacts in separate files. In order to keep consistency, they allow the creation of relations between artifacts, and development environments simulate verisimilitude by presenting related artifacts close to one another.

This dichotomy between single-source and multiple-source approaches is most used in the context of documentation contents – the original concept of LP and Code Annotations are forms of single-source approaches, while TBLP and EP are multiple-source approaches.

2.4.3 Structured Contents

Free-text documents are found most often playing a support role during software development, and other artifacts that are more specialized, such as source code and models, play the central role. From a knowledge capture perspective, each of these different specialized software artifacts implies a particular information structure and therefore is limited to the kinds of subjects that it was designed to convey[15]. On the other hand, they enable a great concision and objectivity when used to convey those particular subjects.

To capture information is also to capture the way it's structured, and two kinds of structure may be considered: the internal structure of an artifact and the structure used to connect and organize artifacts. Capturing the later is as relevant as capturing the artifacts themselves, and plays an important role in assisting information consumers in locating the specific pieces of knowledge they need.

Having explicit relations between different artifacts is also what allows traceability. Relations may connect artifacts that follow each other temporally (predecessor/successor); connect artifacts addressing the same information under different perspectives; or connect artifacts addressing the same information at different abstraction levels, from

[15] Many other kinds of artifacts could be provided as example. Some of them may make more apparent the existence of a structure to which the contents comply, such as structured documents using any dialect of the *eXtended Markup Language* (XML), using the *JavaScript Object Notation* (JSON), or using any other of the many markup languages that are today in common use.

high-level requirements documents to source code. They allow to understand how a set of artifacts has evolved, and which ones have given origin to the others [Meno8]. Some works have shown that traceability can be improved by recovering these links from the artifact's contents, even if the recovered links aren't fully reliable [ACPT01].

2.5 Integrated Environments

Integrated Development Environments (IDEs), such as Eclipse, Netbeans, Rubymine and Visual Studio, join under the same roof several tools required to create software. Although their main focus is on source code, they take a holistic approach to software development, by trying to provide a homogeneous view of software artifacts and the activities in which they are used. They support source code navigation and visualization [SCBR06, SvG05], a quicker expression of the user's intents with features like code-completion [LP88] and refactoring [RBJ97], the overall integration and consistency of artifacts, the maintenance of traceability links, an easier understanding of the software and the processes that surround it, among other benefits. Due to their extensible nature, IDEs tend to evolve into frameworks, which allow one to develop and add to the environment the support for new kinds of tools and artifacts.

Integrated environments have traditionally ran locally on the developer's computer, but several Web-based IDEs already exist and may, in the upcoming years, become as successful as local ones[16].

Meanwhile, a different class of integrated, Web-based, environment has already proven very successful in supporting the activities of developers, despite usually not supporting direct manipulation of source code artifacts: *Software Forges* [REM+09].

The main focus of software forges is not to support directly the creation of source code, although they usually allow navigating and visualizing it. They allow to capture and integrate artifacts as diverse as wiki pages, files from a version-control system, tasks, and milestones, among others. The primary goal of forges is to support open collaboration, making it easy to understand a project in it's several dimensions and contribute to it. The low barrier to entry makes them attractive to non-technical actors. Examples of software forges include SourceForge[17], Trac[18] and GitHub[19].

[16] Notable examples include Could9 (available at `https://c9.io/`) and Coderun Studio (available at `http://www.coderun.com/`).
[17] Available at `http://sourceforge.net`.
[18] Available from `http://trac.edgewall.org/`.
[19] Available at `http://github.com/`.

Software forges present several advantages in terms of simplicity, instant availability of contents and promotion of collaboration between all team members. Web-based IDEs share these benefits and support directly the creation of source code, but traditional IDEs are still more responsive, provide better efficiency for a lot of tasks and support a broader range of tools. In practice, these different environments complement each other, which brings the need for their own integration, which usually happens through the use of plugins like the Eclipse Trac Plugin[20] or the Mylyn GitHub connector[21].

Rational Team Concert is a software-forge-like commercial tool built on the Jazz.net platform that tries to provide an integrated environment for a wide scope of software development artifacts [CHRP03]. Developers use the software to capture relationships between artifacts, and as a communication platform to reach fellow team members and track their activity. It features Web-based and client user interfaces, and integrates with the Eclipse and Visual Studio IDEs.

[20] Available from http://trac-hacks.org/wiki/EclipseTracPlugin.
[21] Available from http://wiki.eclipse.org/EGit/GitHub/UserGuide.

Chapter 3

Software Evolution

Software evolution may be driven by different motivations, reflected on a multitude of different software artifacts, affect the system under development in different ways, and done using different techniques and even perceived differently by distinct team members and stakeholders. This diversity has been captured by taxonomies of software evolution [CHK+01, MBZR03] that confront different approaches and can be used to position the different existing methods and tools.

Research in this area has also identified several forces and challenges [MWD+05, GG08, Men08]. This chapter makes an overview of key notions around the topic of software evolution and looks into *adaptive software* in a closer detail. Figure 3.1 provides a quick overview of the addressed notions. Even though a lot of the issues in this area are addressed in the literature with a focus on source code or the architecture of a software system, they are very often applicable to other software artifacts, including software documentation.

The study of change in the context of software has seen a change in perspective over the years, from the notion of software *maintenance* to that of software *evolution*. This is a shift away from the idea that the changes, so frequently required from software, imply only the replacement of "worn out parts", with the objective of keeping it in function [Swa76, LB85, GG08].

Incompleteness is part of the nature of software systems [GJT09], as they normally can't entirely answer all the needs that could be expected from them, and are asked to deliver only the most important requirements at a given moment. They are frequently the product of the continuous activity of adaptation to change, rather than a static creation, produced to address a given objective that shall remain valid indefinitely. The drift between a system and the requirements that were envisioned for it, is commonly referred to as **software aging**, or *decay* [Par94], and can stem from different causes.

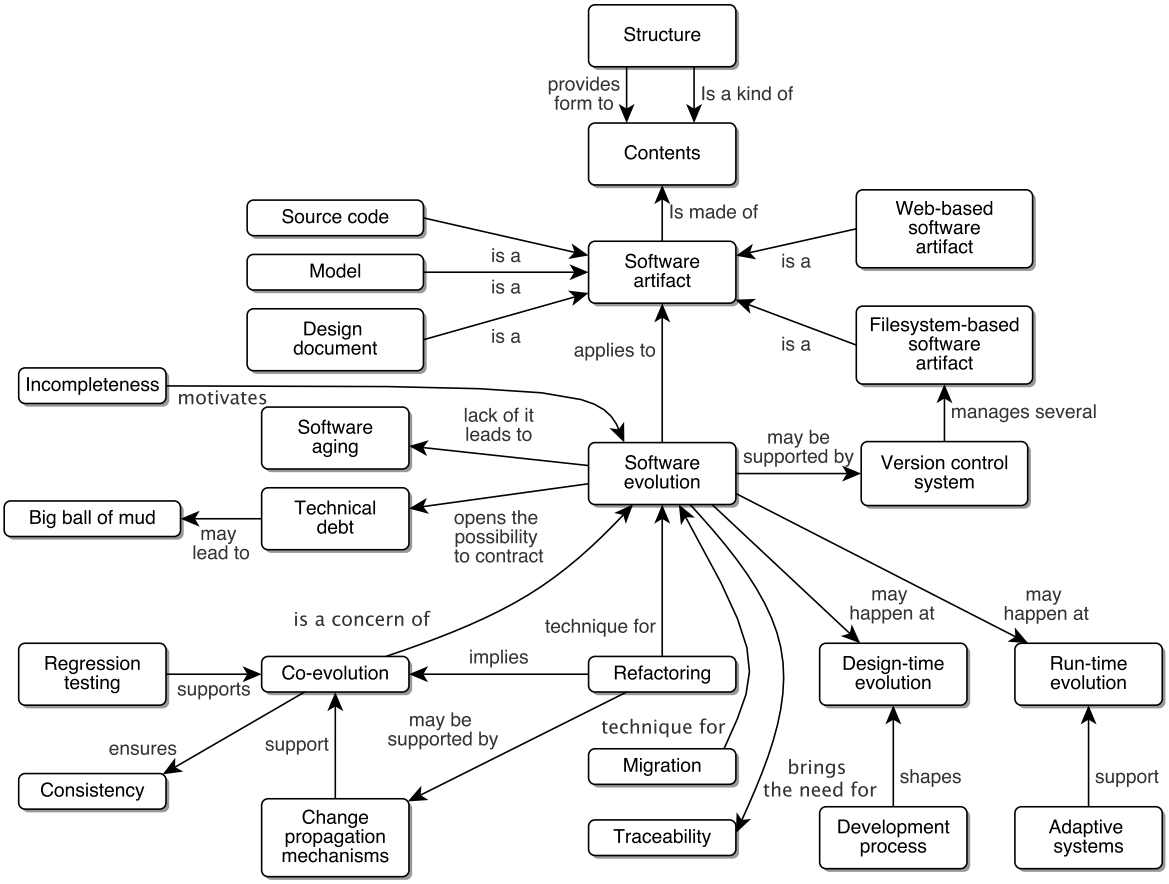


Figure 3.1: Concept map of software evolution topics.

Changes in requirements appear by the hand of the system's stakeholders, ranging from those that appear for business or political reasons, to those triggered by the use of the system itself, which may provide users new understandings of reality and drive further system developments. Not only is reality constantly changing, but so is the stakeholders' understanding of it.

Despite the need to evolve systems, technical issues aren't necessarily always accounted for, resulting in what is frequently denoted as **technical debt** [Cun93]. This term refers to the fact that postponing required changes may imply that subsequent changes become more expensive to implement (i.e., *interest* will have to be paid). While contracting debt may be a deliberate choice to meet the business challenges of a project at a given time, when taken too far leads to a well known pattern, denoted as Big Ball of Mud [FY99], making the evolution costs raise considerably.

3.1 Evolving Artifacts

As introduced before, most software artifacts allow to capture both **contents** and **structure**. These two levels may likewise need to evolve, and they are frequently difficult to change independently of each other.

For instance, when modifying a class hierarchy on object-oriented source code (i.e., structure), developers have also to think about the effects that it will have on the system's behavior beyond the hierarchy itself. Changes will likely be required on the body of those classes too (i.e., the contents). As a second example, it is frequent for common source code fragments to emerge on different parts of a system, which may lead developers to abstract them, and provide them further structure. However, this may be non-trivial, as such code fragments may already be bound to a class hierarchy (i.e., a structure), eventually incompatible with the new one, and that cannot be easily changed without other consequences.

Similar examples can be given for more types of software artifacts and evolution directions. Information may have to be changed because its structure definition has been changed (*top-down* changes), or new structure definitions may be suggested from the information itself (*bottom-up* changes, or *emergent* structure).

The interdependence between parts of an artifact, or between different artifacts, leads to the notion that changes may have non-local impacts, which crosscut different kinds of artifacts and levels of abstraction. This calls for mechanisms that support developers in evolving the software system as a whole – **change propagation mechanisms** [Raj97]. A very common scenario is for source code changes to impact the design and documentation, but many others may be considered. For example, modifying the name of a public method is a change that needs to be propagated to all the source code artifacts that use that method. The need to change different parts of the software together is denoted as **co-evolution** [MBZR03].

The very base of software evolution is incremental change [RG04]. Although some tools provide the mechanisms to keep related artifacts in sync, they rarely allow to keep them **consistent** at all times [SZ01]. **Refactoring** tools [FB99] use change propagation mechanisms to allow developers to modify source code without changing its external behavior. They support the introduction of changes to the internal structure of source code while maintaining its consistency and behavior. Often tools simply provide the mechanisms to assess consistency, so that developers are aware of what still needs to be done to achieve it. For example, although the primary use of compilers is to turn source code into a machine interpretable format, they are very often used to

assess the current syntactic consistency of source code. Some toolsets allow to assess the consistency of documents too, but obviously only for the types of structure that documents allow to express. The domain-agnostic structure of free-text documents, while being one of their greatest strengths, doesn't support assessing if their contents are consistent beyond their layout form. This is a fundamental issue, as the value of documentation is as great as its ability to convey accurate information, but one of the greatest costs, when maintaining documentation for a large system, is ensuring it is in-sync with the artifacts it describes.

Refactoring is one of the techniques that support software evolution. A complementary approach to automating evolution tasks is the use of **migrations** [FCW08, Rub]. Migrations consist in allowing one to express and run a set of transformations to a system's schema and data, making it transition from a version to another. They stem from the need to evolve the domain model (or schema) of a system, which, in turn, brings the need to evolve it's underlying data. The same concept has been used in the context of object-oriented and relational database systems [WE00, RL04].

When it comes to **filesystem-based** artifacts, **version control systems** (VCSs) are extensively used. They allow developers to take *snapshots* of a given set of artifacts throughout the project, so that it's possible to keep track of how they evolve. They also serve as a collaborative platform, allowing developers to deal with new versions of an artifact (or set of artifacts) produced concurrently by different team members. Distributed Version Control Systems (DVCSs) are an interesting variant of VCSs, in that they are especially suited for distributed and disconnected environments, and don't require a version control server to be permanently available to produce new versions.

As introduced in Section 2.4.1, software artifacts can be based on mediums other than files. **Web-based** software artifacts are increasingly popular, even though the wide range of filesystem-based tools are often not suitable to handle them. Integrated development environments have traditionally handled files, but have started supporting more and more Web-based artifacts too [Myl, Mer]. However, some tools are difficult to transition to the Web environment, and the impedance mismatch between artifacts of these two different environments makes it harder to integrate them.

VCSs have proved to be valuable sources of information in the study of how software evolves [Koc05, KCM07, DGLP08, GG08]. They have allowed to identify different ways in which software artifacts evolve, and given hints on how to improve support for the activities that actually take place during software development.

Software evolution may also be classified according to how the software is being used when it takes place [MBZR03]:

> **Design-time evolution** happens when the software is not itself running. It corresponds to the traditional way of evolving software, in which a person or a team introduces changes to the software, normally by producing programming language statements, recompiling it, configuring and deploying it, hence replacing the previous version of the system.
>
> **Run-time evolution** happens when the software being changed is being ran when the change takes place. It usually happens when reflective, or adaptive systems are used. The system alters itself based on inputs of its users or the environment. Concrete techniques for run-time software evolution are addressed in greater detail in the following section as they play an important part in this work.

3.2 Adaptive Software

Adaptive systems are those that can be efficiently molded according to changed circumstances [AG05]. Researchers use the term *adaptive software* in a broad context. Among other possibilities, it may be used to refer specifically to software that a) adapts *itself* (self-adaptive software), to b) software that can be easily changed to accommodate new requirements, or to c) software able to satisfy many different user or market needs [AG05]. The present section doesn't try to address all these facets, and focuses on the third use of the term, which is the most meaningful one for the context of this work. A quick overview of this area of study and, in particular, of the topics described in this section is depicted in Figure 3.2.

One way of creating an adaptive system is through a **meta-architecture**: one in which the program manipulates itself as if it were data. Meta-architectures usually describe the system's domain, or a part of it, by establishing different levels of (meta-)data that comply to each other.

Meta-modeling can be used to create such descriptions. Dynamic meta-modeling approaches, such as those using the Adaptive Object-Model (AOM) architectural design pattern [YBJ01], allow systems to be adapted at runtime – the system interprets a high-level description of the domain, and adapts its behavior to any changes introduced to that description.

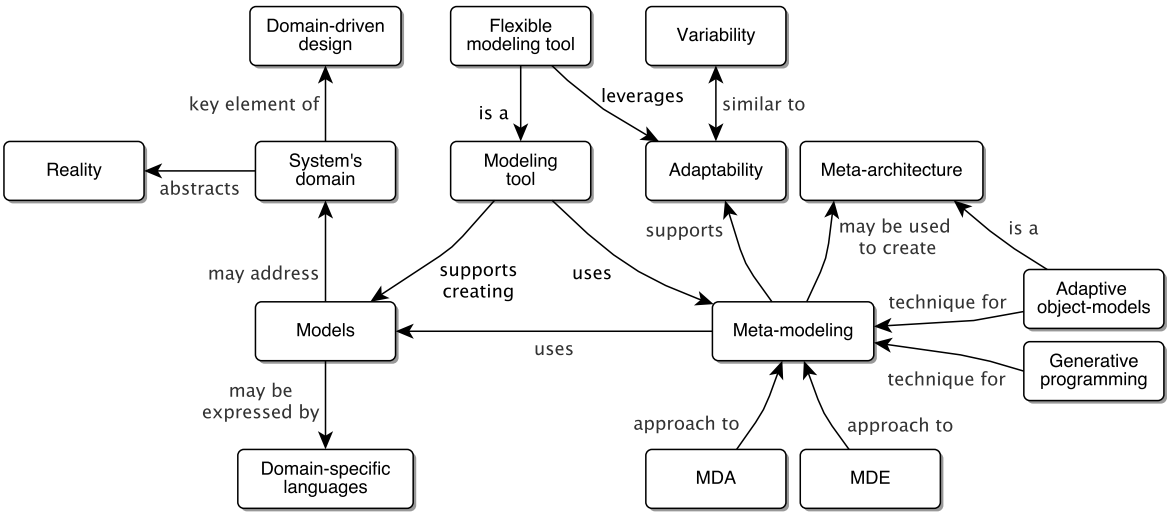


Figure 3.2: Concept map of adaptive software topics.

An alternative to dynamic approaches is **generative programming**: the high-level description of a system is used to automatically create code that can be executed, or a code skeleton that will be further completed by the developers. Adaptability is thus introduced at compile-time, requiring a full generation/compilation cycle.

These notions are explored in the following sections to a greater extent.

3.2.1 Domain-driven Design

The purpose of developing software is, ultimately, to fulfill its users needs, but to actually understand these needs developers may have to delve into those user's knowledge domain, with all that it entails. **Domain-driven design** (DDD) is an approach to the development of software that strives to closely connect the implementation to the business domain. As Eric Evans puts it, in the book *Domain-Driven Design* [Eva03]:

> *"To create software that is valuably involved in users' activities, a development team must bring to bear a body of knowledge related to those activities. The breadth of knowledge required can be daunting. The volume and complexity of information can be overwhelming. Models are tools for grappling with this overload. A model is a selectively simplified and consciously structured form of knowledge. An appropriate model makes sense of information and focuses it on a problem."*

This approach fosters creativity and collaboration between domain experts and developers, making models the center artifacts and part of the language used during those activities. These **models** are not an accurate description of **reality**, but rather

abstractions, representing of the **system's domain** with the very focused goal of feeding the development process.

3.2.2 Adaptability and Variability

The general definition of **adaptive systems** as those whose behavior can be easily changed according to new realities and needs meets the work of several authors, although establishing what the term *adaptability* means hasn't always been consensual [AG05].

Variability is not an exact equivalent to adaptability, although they are very close concepts, that overlap. The notion of variability was born on the area of Software Product Lines (SPLs), and has been defined as a property that allows changing or configuring a system so that it may be used in different contexts [vGBS01]. Systems provide (and constraint) variability through variation points, which are the points of the software that can be changed, and that define to which degree it can be customized.

The key difference is that when addressing the topic of adaptability there is a whole area of the system that can be changed according to the developer's (or user's) needs, and when addressing variability, the focus is on a constrained set of customizations points.

3.2.3 Meta-modeling

Higher-level programming languages have increasingly supported developers on focusing on the design of the software being built, rather that its implementation details [Sch06]. **Model-driven engineering** (MDE) continues this trend, with the objective to further reduce the gap between specification and implementation artifacts, by creating models that abstract several facets of software development. It provides benefits such as increased reuse, fewer bugs, shorter time-to-market and systems that are simpler to understand [RFBO01].

But more than supporting the creation of models, MDE is an approach to **meta-modeling**: it supports the use of models to specify other models – it comprises the *analysis, construction and development of the frames, rules and constraints to modeling a predefined class of problems* [SV06].

A wide range of modeling languages exist, from textual **domain specific languages** (DSLs) built for a very specific purpose, to graphical languages like the **Unified Modeling Language** (UML). UML is currently one of the most widespread modeling

languages. It can be used to specify software systems through a graphical notation, allowing to express elements such as classes, actors, activities and components, among others. It is based on the infrastructure provided by the Meta Object Facility (MOF) [OMG], which defines a meta-modeling architecture consisting of four modeling levels, each conforming to the one above – M0, M1, M2 and M3, with M0 corresponding to the *data*, M1 to the *model*, M2 to the *meta-model* and M3 to the *meta-meta-model*, which is compliant with itself. An example of these layers of abstraction is depicted in Figure 3.3.

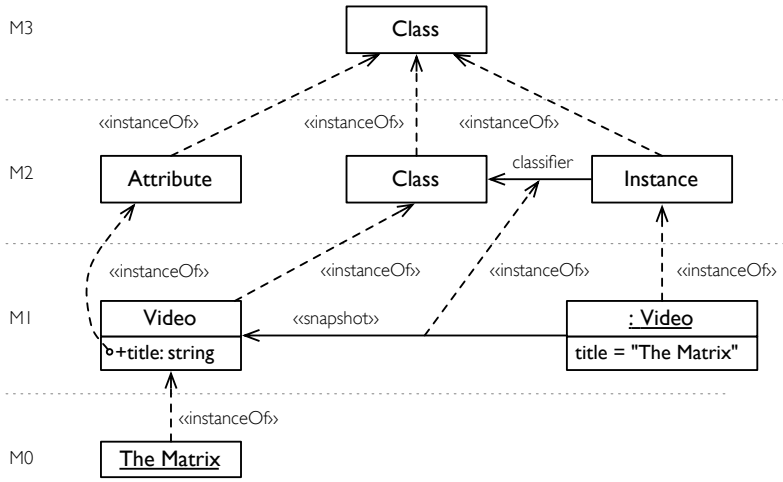


Figure 3.3: MOF's abstraction layers, adapted from *Adaptive Object-Models: a Research Roadmap* [FCAF10].

Due to the high level of abstraction provided by MDE approaches, they can be wielded to express any desired topic, and to structure information according to an intended domain.

3.2.4 Flexible Modeling

The kinds of artifacts and techniques described in the *Software Documentation* chapter (Sections 2.2 and 2.3) allow different levels of expressiveness. Free-text documents allow to convey information in a wide range of domains but usually implies a certain ambiguity and verbosity. On the other end of the spectrum, source code statements can be quite unambiguous and terse, but they can only convey information on the domain of computations. A lot of software artifacts lie somewhere between these two extremes.

Being able to extensively and accurately capture their knowledge using only source code and the occasional comments can still leave a lot to be desired, in spite of all the

advances in what concerns programming, domain-specific languages and modeling [Wir08].

Even experienced developers start by recording ideas in an unstructured and informal way, and only gradually are able to create more specialized software artifacts, like models and source code. A particularly important challenge is to support the process of moving back and forth between these different degrees of structure. This subject has gathered some attention, as more researchers acknowledge its importance, and some works have been published under the topic of *flexible modeling tools* [OvdHS09, KOvdHS10, OvdHS+10a, OvdHS+11].

We will briefly examine two platforms that use *flexible modeling* approaches: *Architects' Workbench* and *Business insight toolkit*.

Architects' Workbench

The Architects' Workbench (AWB) is an Eclipse-based tool that has the general goal of supporting the process of architectural thinking and modeling. Its objective is *"to balance formalism and freedom, while helping [architects] transform unstructured information into sufficiently formal work products"*[ABK+06]. When using AWB, software architects usually begin with a combination of informal free-text documents and notes from meetings with stakeholders, and create model elements directly from the text. Architects can then navigate between these elements and the text, bidirectionally.

To support creative thinking, refinement of the model can be deferred to a later time. It can be refactored as understanding of the domain improves, either through form-based user-interface or through customized diagram editors. As explained by Kimelman et al [KH11]:

> *"[...] even just that small amount of structure requires some decision by the user concerning the visual structure or appearance of each fragment prior to it being placed onto the canvas, and it requires some thought by the user to accomplish the necessary keyboard actions, mouse actions, or gestures. That "second order" thought is an impediment – it disrupts the primary train of thought, it is an obstruction that impedes the flow of thoughts and ideas into artifacts, and it can stifle creativity... even for many experts."*

Business Insight Toolkit

The Business Insight Toolkit (BITKit) is a standalone modeling tool that takes inspiration from AWB [OBA+09]. Developers use BITKit to sketch diagrams without a

formally defined semantics, and can afterwards map them to new or existing elements of a domain model. Like AWB, it has a forgiving approach to the creation of models: instead of using the meta-model as a straitjacket that establishes what can be expressed at the model level, it uses it as a guiding aid, which supports the creation of the model but allows developers to deviate when needed.

Comparing to AWB, BITKit doesn't take text as a starting point, but provides better diagramming tools and allows more flexibility at the meta-model level. While AWB needs the system to be reconfigured to introduce changes to the meta-model, BITKit allows to derive the meta-model from the model data, effectively providing guidance in both directions – the creation of the model can be driven by the existing meta-model, and the creation of the meta-model can be driven by the existing model [OBS+10].

3.2.5 Adaptive Object-Models

The Adaptive Object-Model (AOM) is an architectural design pattern[1] that allows end-users to manipulate the domain model underlying the software system. It may provide this kind of flexibility for the entire domain model or for a selected part of it. This architecture makes extensive use of notions from object-orientation and meta-modeling, supporting reflection and runtime adaptivity, and often relying on domain-specific languages (DSLs). It can be said that it is an architecture focused on *embracing change* of the system's problem domain.

Developers sometimes converge to this architectural design pattern by systematically improving their reuse strategies and searching for higher levels of abstraction in object-oriented designs. It frequently emerges by making domain-related data structures into parameters of the system, which can be configured according to user's needs. Some parts of the system are this way turned into an interpreter, and the system's behavior is decided at run-time from the provided parameters. As more domain elements are parameterized, a *model* starts taking shape, and changing such model, makes the system follow a different domain.

This kind of systems has been documented through design patterns, with the objective of creating a pattern language for AOMs. Figure 3.4 was adapted from other works [WYWJ07, WYW07, Fer10] and depicts a pattern-map of that language.

Without going into detail about the objective of each of them, the following four

[1] A *pattern* is a recognized solution to a recurring problem that can be described in terms of a context of applicability and a set of forces that shape the solution. The notion of *pattern* is described in greater detail in the *Patterns Catalog* (Chapter 5).

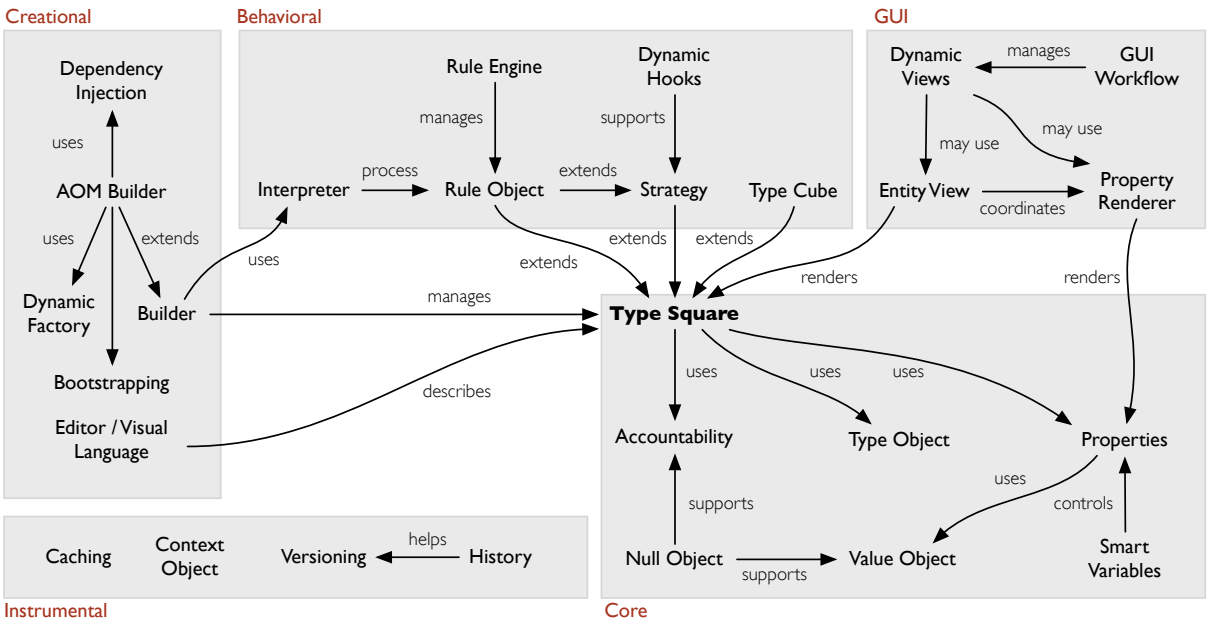


Figure 3.4: Pattern map of the adaptive object-models patterns as defined by previous works.

design patterns are the most important to understand the structural aspects of an AOM:

Type Object – Decouples instances from their classes so that those classes can be implemented as instances of a class. Type Object allows new "classes" to be created dynamically at runtime, lets a system provide its own type-checking rules, and can lead to simpler, smaller systems [JW97, YBJ01].

Property – This design pattern gives a different solution to class attributes. Instead of being directly created as several class variables, attributes are kept in a collection, and stored as a single class variable. This makes it possible for different instances, of the same class, to have different attributes [YFRT98, YBJ01].

Type Square – The combined application of the Type Object and Property result in the Type Square [YBJ01]. Its name comes from the resulting layout when represented in class diagram, with the classes *Entity*, *Entity Type*, *Attribute* and *Attribute Type*.

Accountability – This design pattern is used to represent different relations between parties [Fow97, Ars00], using an *AccountabilityType* to distinguish between different kinds of relation.

Chapter 4

Research Problem and Strategy

This work addresses knowledge capture and acquisition in the context of software development with the overarching goal of improving software documentation. To do this it focuses on the objects of capture and acquisition activities – software artifacts. In particular, it focuses on software artifacts that may be used as documentation and the approaches and tools used to handle them. In Chapters 2 and 3 are identified relevant documentation approaches and tools and considerations are made about their designs, with a very special attention to the developers' need to evolve knowledge and the concrete artifacts that capture it.

This chapter puts the research into a more specific context (Section 4.1), motivates and describes the research problems (Sections 4.2 and 4.3), presents the thesis statement and its decomposition into specific research issues (Sections 4.4 and 4.5) and describes the outcomes and the validation that are goals of this work (Sections 4.6 and 4.7).

4.1 Context Overview

Software artifacts are often captured as standalone files, which are sometimes managed by integrated development environments; other times they are captured using web-based systems, such as software forges. These platforms complement each other – while integrated development environments focus primarily on the creation of source code, software forges have been difficult to match in their support for collaboration and ability to cross-reference several kinds of artifacts.

But regardless of the medium, artifacts are a form of captured (i.e., recorded) knowledge. While new knowledge may easily be created and evolved in the minds of the project's team members, artifacts hardly capture all of it. Developers are not always

able to share the commonalities and regularities that they may identify in free-text information; it is often difficult to maintain such information consistent; and it's a real challenge to find classification schemes that will hold during the project's lifetime, and can help people with different goals to find the information they need.

Free-text Information

Information producers are fond of free-text documents for their flexibility and freedom from rigid structures. Narrative contents naturally fit free-text and it can be used to convey ideas pleasantly and effectively. A free-text document can be much easier to create than other artifacts and it can address a wide scope of topics and be made to be enticing to the particular audience that it targets.

But semantically richer representations also have their place. When authors know the model to which their contents obey, they may wish to explicitly capture and communicate it.

Structure and Models

Structure refers to a set of different information fragments arranged together in a determined way and many times in function of the subjects that the information addresses. In this sense, models are structured information that has the specific goal of simplifying and representing a complex reality. Models are used to explicitly represent the important details of a given domain, relinquishing unimportant ones to the background.

All the information that we could abstract in some way may be said to possess an underlying model that could be made explicit. The more unambiguous and crystallized a certain knowledge is, the easier it may be recorded into a concrete model. The key benefit in the use of models is that, by abstracting and explicitly representing information, computers may be made to process it effectively, and humans can more easily perceive it, as they can understand some information (model) in terms of another (meta-model).

Different modeling levels may be considered. As **models** establish structure for bare information, **meta-models** can be said to establish structure for models themselves. Models and meta-models may be used for several purposes in the process of developing software, and different techniques may be employed, ranging from generative to dynamic ones [YBJ01].

Software Artifacts

Software artifacts are abstractions derived from the knowledge of developers – they are the *things* that software developers work with, like *documents*, *source code*, *domain models*, etc.

This suggests that all software artifacts may be regarded as having an *identity* – what allows to say that two artifacts are different ones – and a *type* – which reflects the kind of structure they possess and may have in common with other artifacts (i.e., they may be of the same or of different types). These common structures may be seen as an underlying model of the artifacts. One may climb one step further in the abstraction ladder and find a meta-model, which may be common to a wide set of artifact types.

Some examples of such different model levels are depicted in Figure 4.1, showing a few example software artifacts (wiki-based free-text documents, source code and software-forge-based artifacts) and different levels at which it is possible to regard them. These different levels are strongly correlated with how developers deal with software artifacts, in the sense that few toolsets and kinds of artifacts allow modifications to be made at multiple levels.

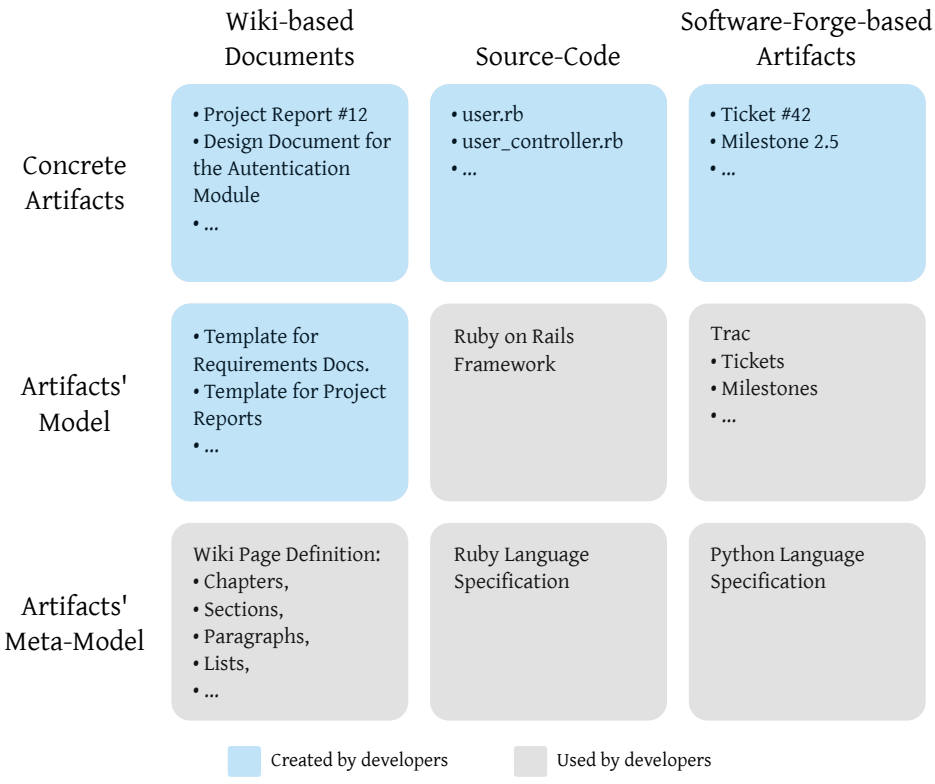


Figure 4.1: A view of the different model levels at which some example software artifacts may be seen.

Some of these levels are able to receive arbitrary modifications even though those are impractical or difficult to do in reality. Software forges usually try to keep the core of the system simple, supporting only a set of software artifacts considered by its creators to be almost of universal use, and rely on plugins to support additional kinds of artifacts. Developing and maintaining such plugins requires knowledge of the design and extensibility mechanisms offered by the software forge, and it entails an effort that may be difficult to justify if the plugin is used by only a few projects.

4.2 Motivational Example

Take as an example a software project that will soon be started by John and his team mates. This section will use this example to illustrate three concerns: expressing structured contents, maintaining the consistency of textual contents, and classifying textual contents so that they can be easily reached.

◇ ◇ ◇

John and his team have started the new project by conducting some exploratory meetings with prospective customers. They have created a series of informal pages in the project's wiki, with the results of the meetings. These contain different perspectives that the team will distill into a single vision, and into a concrete software system. More specifically, the team will want to identify the problem domain that underlies this body of information, and create more specialized artifacts – such as user stories, tasks, models and source code – and in this process give the information a more concrete and unambiguous form. As John captures contents and goes through existing ones he starts identifying some commonalities between them. He speculates that a part of the contents may become tasks and other parts may be referring to entities of the problem domain, but he is not sure yet. Despite that, he tries to capture the contents as these more specialized kinds of artifact, which could make common information structures more apparent, but this diverts him from his main train of thought. Hence, he chooses to take only textual notes and organizes them to match his current understanding to the extent that is possible. His understanding of these contents is still to speculative and changing too frequently, so he finds that it is premature to structure information more thoroughly at this time.

◇ ◇ ◇

After some months of work, the development of the system is well under way and

the team has produced many more artifacts. Among them there are now some free-text documents describing the system's requirements and making some architecture considerations, as well as tasks, models and source code.

But the team's knowledge keeps evolving. John has just updated a page in the team's wiki that, among other topics, describes how the User Accounts work. Although he doesn't have a full view of all the information in the wiki, there are several pages that refer to User Accounts. On updating that document, John has unknowingly introduced an inconsistency with at least two other pages: a recipe for creating pluggable authentication backends, and an overview of the system's architecture.

◇ ◇ ◇

Having now a considerable amount of free-text documents, the team is often faced with the challenge of finding the information they need to perform a given task. To ease browsing, the wiki's entry page is arranged as a table of contents, with each of its entries referring to a specific area of the problem domain.

John is developing a new authentication backend; he is searching the wiki pages for how to extend the authentication module and he is coming to the conclusion that organizing contents following exclusively concepts of the problem domain works for a lot of cases, but technical information is still sometimes difficult to reach. To make these wiki pages easier to find, John extends the table of contents with technical topics, although he knows that, because of it, the table of contents page will become more difficult to maintain and to navigate.

4.3 Concerns

The examples described in the previous section help to illustrate the three key concerns that this research tries to address – *expression of information structure*, *consistency maintenance* and *classification of the contents*.

C1. Expression of information structure

The artifacts created by a team can record diverse kinds of information, like requirements, architecture and the process of software development itself. Each type of artifact by its very nature implies a different kind of domain structure.

While some information may first be captured as free-text documents, to become more useful it will eventually need to become more specialized software artifacts. This

move implies a trade-off, between the ease of evolving unstructured information such as free-text, and the advantages of making the information's underlying structure explicit by creating artifacts such as tasks or models. Namely, capturing information's structure explicitly enables us to share it with others and for them to understand the information in terms of that structure. The creation of specialized artifacts also enables the use of specific tools, which are more effective in processing those specific kinds of contents.

But the recognition of the underlying structure of a body of information is an incremental process, and one that tools don't necessarily support. Developers sometimes avoid the effort of capturing free-text documents and only capture more specialized and formal artifacts, but they tend to do so only when they have the confidence of having found the right abstraction (i.e., the right kind of artifact) for a specific piece of information, and thus inevitably delay knowledge capture.

C2. Consistency maintenance

The same topic may be described in different facets, by different free-text documents. But the connections between these documents is usually captured weakly as free-text themselves, making the maintenance of consistency between the different documents depend on an expensive process of continuous review.

If topics are addressed by as few documents as possible, the number of dependencies will be lower, which will ease maintaining consistency. But the most effective documentation is the one tailored to support specific tasks and audiences, which usually requires documents to span several topics.

The larger a body of information is, the more costly it is to maintain its consistency but, on the other hand, we want to capture as much information as can be useful to the team in the future.

C3. Classification of the contents

Two predominant modes are used to seek contents – browsing or using a text search. Even though search boxes are nowadays prevalent in software environments, some studies suggest that browsing is still often preferred over a text search as it enables user orientation and the opportunity to refine information needs and gain context [KB03, TAAK04].

To enable effective browsing of free-text contents, they need to be classified and accessible through an index, but it's usually difficult to create a single classification scheme that can be effective to find any free-text document, and be used by the whole team. Different team members will have different prior knowledge on any given subject, which means that they may need different starting points to seek information on a topic. But creating and maintaining a classification scheme with comprehensive starting points can take a lot of the team's energy.

By creating a semantically rich classification scheme, the reader can be better guided from his starting point to the information he seeks, but again, building such an elaborate index implies a higher maintenance cost that a team may not find worthwhile.

4.4 Thesis Statement

Although many developers create software systems to manage information (often denoted as *information systems*), they don't usually see the primary means of their own environment and the many products of their work as information that they have to manage. We recognize that to make the most of the information captured during software development, it must be regarded as a primary source of knowledge. This research thus started with a question – *what lacks current documentation approaches to make the most of the information captured during software development?* – and proposes an approach for software documentation (Chapter 6) that it names *Adaptive Software Artifacts*.

In brief, the author's thesis is that:

> *Capturing software knowledge with the Adaptive Software Artifacts approach makes information easier to be consumed, created and evolved, especially in the context of medium-to-large projects.*

More specifically, it is our goal to research the effects of the Adaptive Software Artifacts approach for software documentation, when *using* software documentation and when *creating or changing* it, focusing on the three key concerns introduced in Section 4.3 – a) expression of information structure; b) consistency maintenance; and c) classification of the contents. The objective is to see if the Adaptive Software Artifacts approach enables to share information structure more efficiently and to improve free-

text documents by allowing to maintain their consistency and classification with more economy and quality.

This explanation uses terms that may be subject to different interpretations and therefore should be clarified:

What should be understood by *contents* and *information*?
Contents and *information* are used interchangeably in this dissertation to refer to a manifestation of knowledge after it has been recorded in a (usually digital) medium.

What should be understood by *software artifacts*?
A recorded, identifiable, piece of knowledge about a software project and/or that results directly from the act of developing software.

What should be understood by *using* or *consuming* information?
Acquiring information from a medium, usually with the goal of actively employing it for some purpose. In other words, reconstructing mental models, transforming *information* back to *knowledge*.

What should be understood by *creating* and *evolving* information?
Creating information is to capture knowledge into a medium from existing mental models. To change or to add to that information (e.g., so as to capture different or additional knowledge) is to *evolve* it beyond its current complexion.

What should be understood by a *medium-to-large* software project?
A software development endeavor made non-trivial by the quantity and complexity of information involved, which often correlates with a larger number of team members.

What should be understood by *information structure*?
Structure in this context refers to the composition of different information fragments in a determined way, often in terms of the subjects that the information addresses.

What should be understood by *consistency*?
The quality of being in agreement with something else. Maintaining the consistency of two related pieces of information is to ensure they don't convey contradictory ideas.

What should be understood by *classification*?
The organization of information into groups or classes, according to their common natures or subjects.

4.5 Specific Research Issues

The thesis and the concerns introduced so far may be broken down into the more specific issues that we will address next and that were used to drive the validation of the approach. Together, these issues establish a direction for this and future research on software documentation and on the Adaptive Software Artifacts approach in particular.

While issue I1 focuses on knowledge acquisition (i.e., information *consumption*), I2 focuses on knowledge capture (i.e., information *creation* and *evolution*). Both of these issues are further decomposed into the three concerns described in Section 4.3.

I1. Efficiency of Knowledge Acquisition.
Do developers spend less *time* acquiring knowledge from the contents?

I1.1. Efficiency of acquiring information structure.
Do developers spend less *time* understanding the contents?

I1.2. Consistency of the contents.
Are resulting contents more *consistent*?

I1.3. Quality of the classification scheme.
Do developers spend less *time* searching for the contents they need?

I2. Efficiency of Knowledge Capture.
Do developers spend less *time* capturing contents?

I2.1. Efficiency of expressing information structure.
Do developers spend less *time* capturing the contents?

I2.2. Economy of consistency maintenance.
Do developers spend less *time* doing consistency maintenance?

I2.3. Economy of classification scheme maintenance.
Do developers spend less *time* maintaining a classification scheme?

4.6 Research Outcomes

This research produced four main outcomes in the path of pursuing answers to the above issues. These contributions are depicted in Figure 4.2 and include, a) a patterns catalog, b) an approach to software documentation, c) a reference architecture and d) a statistical experiment.

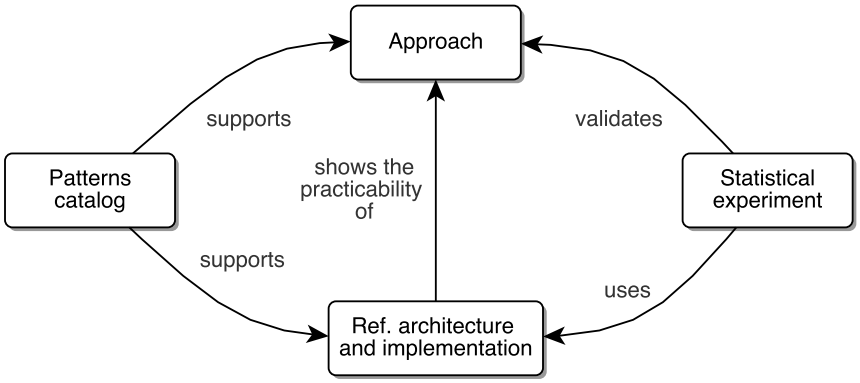


Figure 4.2: Concept map of the research outcomes and of how they relate.

The creation of the **patterns catalog** was done throughout the research, by mining the patterns from the literature and the authors' personal experience in the area. They served to formalize several good practices and designs surrounding software documentation, information classification, flexible modeling tools and adaptive object-models. The resulting patterns are presented in Chapter 5.

A new software documentation **approach** was defined – the Adaptive Software Artifacts approach – that embodies some of the good practices and designs that were documented as patterns and tries to address the key concerns of this research. The approach is described in Chapter 6.

A software documentation tool was engineered as a plugin for a Web-based environment for software development. It uses some of the more technical solutions that were described as patterns, and proves the concept of the approach. The design of the plugin is detailed in Chapter 7 and is available for use as a **reference architecture and implementation** by other developers.

Furthermore, the plugin was used in a user-study with the goal of experimentally validating some of the benefits of the approach. The design of the **statistical experiment** and its results are detailed in Chapter 8.

4.7 Validation Methods

It may be argued that research in the social sciences is necessarily *qualitative*, as the data that it produces is mostly non-numeric in nature, even if the conclusions drawn from such data may sometimes be *quantified*. However, this view is not consensual – Goertz and Mahoney go to the extent of considering that these views stem from two fundamentally different cultures behind the dichotomy of quantitative and qualitative research, and that the first is mainly based in inferential statistics, whereas qualitative research is grounded in logic and set theory [GM12, p. 2]. Throughout this thesis we will often refer to the collected data as quantitative despite its origins on human activity.

Software engineering research goes beyond concrete implementations and tools and it too needs to consider the whole context in which software development takes place, and the human and social aspects that underly it. Given the amount of possibilities that this implies, software engineering is an area where applying quantitative methods is often difficult.

One of the approaches used by the author to validate his thesis consists of a statistical experiment, pulling this work closer to what is often referred to as quantitative research. Statistical experiments measure the causal link between an independent variable and the phenomenon under study by ensuring that the remaining variables are *controlled* – that is, that they are kept close to constant using *repetition*, *randomized assignment* and *averaging* [USE, DV99]. They generate quantitative data that is the focus of a statistical analysis.

As suggested before, experiments are more difficult to conduct in software engineering than in other areas in which the independent variables are easier to control. They are, however, the best way to assess cause and effect relationships. The method used in this work can be denoted as a *quasi*-experiment, implying that there is not a full control over some independent variables, but otherwise follow the same rules as traditional experiments [CJMS10]. Chapter 8 fully details how the experiment was designed and executed. It focused specifically on those issues related with knowledge acquisition, rather than trying to address all of them as they were introduced in Section 4.5.

4.8 Summary

Software development tools support the evolution of artifacts to some extent, but they normally assume that artifacts have a fixed format.

Free-text documents don't force any domain structure, and for that reason, are very flexible when capturing and evolving contents. On the other hand, the lack of domain structure makes them less expressive and thus more laborious to produce and harder to automatically process.

Other software artifacts depend on rich domain-oriented structures and thus support a greater expressiveness. However, this means their creation and evolution is bound to structural constraints, which may be too strict and make it impossible to mold the artifacts to new realities.

Artifacts that could be easily evolved throughout the project's lifetime, in what concerns both their contents and structure would bring benefits to knowledge capture activities. Information usually first appears as informal and only gradually becomes more structured and is captured into richer artifacts. A good example is how requirements are often first captured as descriptive free-text documents and, only afterwards, materialize as models, tasks, and source code, among others.

The thesis introduced in this chapter looks into the acquisition and the capture of knowledge through the prism of three main concerns – a) the expression of information structure, b) the maintenance of consistency and c) the classification of the contents. This translates into the eight main research issues and sub-issues that are used in the validation of this work.

The four main outcomes of this research complement each other: the *reference architecture and implementation* shows the practicability of the defined *approach* and they both are supported by the solutions described in the *patterns catalog*; the *statistical experiment* uses the *reference architecture and implementation* in a user-study to validate some of the benefits of the *approach*.

Chapter 5

Patterns Catalog

We have used *patterns* to document key solutions in topics that have shown to be of interest to this research. Some of these solutions were identified by the author while others had already been identified before, but had not been captured as consistently using patterns.

Before delving into the patterns themselves (Sections 5.4 to 5.7), we will explain the essence of what patterns are and their role in this research (Section 5.1), what pattern form we have used (Section 5.2) and which are the main topics that these patterns aim to address (Section 5.3).

5.1 Patterns in Research

Patterns are *general reusable solutions to commonly occurring problems within given contexts*. The notion was created in the field of architecture by Christopher Alexander [Ale77, Ale79], who originally defined the concept as:

> *"[...] a three-part rule, which expresses a relation between a certain context, a problem, and a solution.*
> *As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.*
> *As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant."*

In spite of its roots, this notion can be (and has been) applied to other domains in which solutions need to be *designed*, given a specific problem and context. Experts in

many areas, including software engineering, think in terms of such problem-solution pairs when faced with new challenges [BMR+96].

Patterns are not *invented*. They distill knowledge in a given domain and, in doing so, allow that knowledge to be reused and provide a common vocabulary for understanding and communicating design principles [GHJV95]. They are highly reusable because the solutions that they provide are abstract – they may be applied *a million times over, without ever doing it the same way twice* [Ale77].

When organized in a collection, patterns can provide a useful repository of knowledge. In its simplest form, a set of patterns organized according to a similar context or purpose can be said to be a *patterns catalog*. The patterns in a catalog don't necessarily have to work together in addressing a domain. When the relations between them are made explicit, when they consider the impact that a given solution has on other solution spaces and when they address the domain at different levels, they may be said to constitute a *pattern language*. The use of a pattern language thus implies a holistic approach to a domain, as it guides designers through the problems across that domain, suggesting which patterns should be used in each case [BHS07].

As mentioned previously, the goal of patterns is not to present new ideas, but to represent what *is*. In other words, they formalize empirical observations by expressing the invariants of problem and solution spaces, and they provide abstractions that support solutions for those problems. As advocated by Kohls and Panke [KP10], patterns can be seen as specific kinds of theories – good-practices theories – that are identified through a process very similar to that of scientific discovery. The patterns community calls this process *pattern mining* – the *discovery of nuggets of wisdoms*, as so expressively described by Kohls and Panke. The methods that lead to this discovery are based on inductive inference, which is common on qualitative research, and the validity of a pattern is supported by its empirical content and known uses.

The reference architecture and implementation described in Chapter 7 is possibly the most tangible part of this research from a software engineering standpoint, but the goal of this work goes beyond concrete implementations and tools. This patterns catalog embodies knowledge for *designing* an environment that supports the Adaptive Software Artifacts approach. Despite this, it doesn't mean that the patterns are specific to this approach – we expect most of them to be useful when creating other kinds of systems for documenting software.

5.2 Pattern Form

Authors tend to find their own form when writing patterns, but often draw inspiration from forms that, for one reason or another, have become more popular. Three of the most influential pattern forms are the *Alexandrian form*, the *GoF form* and the *POSA form*. These forms differ in the order and structure of the contents; some of them are more narrative and others more explicitly structured (i.e., make more extensive use of section headings and lists). The Alexandrian form, as the name suggests, was used by Christopher Alexandre, namely in his book *A Pattern Language* [Ale77]. It is a very narrative form, using few section headings and relying considerably on visual formatting elements to cue on the organization of the narrative. Almost on the opposite end of the spectrum, there's the GoF form, used in the influential Gang of Four book[1] [GHJV95]. This form is very structured, and a strong departure from the Alexandrian form, breaking the patterns in several sections. Finally, the form followed by the *Pattern Oriented Software Architecture* book series (i.e., the POSA form) is, like the GoF form, very structured, but the narrative follows an order somewhat closer to the Alexandrian form [Fow06].

The form of the patterns that you can read on this chapter was strongly influenced by the POSA and Alexandrian forms. The headings are mostly inspired in the POSA form, but only the most important ones were used. Whenever appropriate, we have replaced some of the POSA section headings by formatting cues. This supports a more narrative style than the original POSA form, and allows encompassing patterns of different levels of detail. The patterns were broken down in the following parts:

Name. The name by which the pattern is known. The pattern name conveys the main idea of the underlying solution.

Figure. A figure that visually conveys the pattern. Optional – a figure was not included for every pattern.

Context. A description of the setting in which the pattern occurs. Often references patterns of a higher abstraction level, which set the context for the pattern being described.

Example. A concrete example of the problem addressed by the pattern. Optional – this section was not used for all the patterns.

[1] Commonly called this way due to its four authors. The book's published name is *Design Patterns: Elements of Reusable Object-Oriented Software*.

Problem. Starts with a one-sentence problem statement that captures the main issue that the pattern tries to address, and that is visually highlighted through a different formatting. The remaining of the section details the problem and the (often opposing) forces that shape the solution.

Solution. Like the *problem* section, this section starts with a visually highlighted one-sentence statement, which tries to capture the key idea of the solution. It then goes on to explain with more detail how the solution can actually be put into practice, and what are the positive and negative consequences of applying it. This section may also include the multiple variants of the solution, whenever they exist.

Example Resolved. This section describes how the particular problem scenario described in the *Example* section can be concretely solved by the solution. Optional – this section was not used for all the patterns, and only appears when an *example* section also exists.

Known Uses. This section provides examples of where the pattern may be observed in practice.

Related Patterns. Other patterns, which may be used synergistically with the one being described. While the *context* section often mentions patterns of a higher abstraction level, this section covers all related patterns and givens more focus to patterns of a lower abstraction level.

5.3 Catalog Overview

The patterns addressed in this chapter can be grouped into different sets. They play different parts in the approach (Chapter 6) and in the reference architecture and implementation (Chapter 7).

Patterns of Consistent Software Documentation. Documentation is an important part of the captured knowledge of a software project, providing a flexible and effective way of recording informal contents. However, maintaining it consistent requires a considerable effort. Existing solutions encompass different tools and approaches that support the process of creating, evolving and using free-text documents and other artifacts derived from the software development process. We have identified key problems and solutions for documentation consistency based on existing literature and

personal expertise. In concrete, four distinct patterns and their relations were identified – INFORMATION PROXIMITY (p. 66), CO-EVOLUTION (p. 71), DOMAIN-STRUCTURED INFORMATION (p. 74) and INTEGRATED ENVIRONMENT (p. 76) [CFFA09b].

Patterns of Information Classification. Providing efficient access to information can be approached in different ways, but ultimately implies the creation of an INDEX (p. 82), represented with an indexing language, like a TAXONOMY (p. 85), a THESAURUS (p. 88) an ONTOLOGY (p. 91) or a FOLKSONOMY (p. 94). Each of these languages strikes a different balance between the effort to create and maintain the index, the effectiveness of knowledge capture, the guidance that readers can get, and how efficiently they can get it. Furthermore, THESAURI and ONTOLOGIES rely on the use of a the CONTROLLED VOCABULARY (p. 97) to disambiguate the meaning of terms [CA11] .

Patterns of Flexible Modeling Tools. The benefits of using models have long been acknowledged by research and industry, but in practice the use of modeling tools often implies an unreasonable effort or confines itself to points in the project lifetime when the requirements and/or design is well understood. Free-form tools like whiteboards and textual documents fill information capture needs during the rest of the time – they pose a lower barrier for adoption and enable users to capture their flow of thought with fewer constraints than a formal modeling tool would allow. Flexible Modeling Tools try to provide a compromise of both approaches. The works discussed at the FlexiTools workshop series represent an interesting body of knowledge, covering different issues and approaches for this class of tools, and was one of the main sources for mining these patterns – USER-CRAFTED STATIC META-MODEL (p. 103), MODEL CO-EVOLUTION (p. 104), META-MODELING BY EXAMPLE (p. 106), FORMALIZATION (p. 108), LINKED MODELS (p. 110) and AUGMENTED MODELS (p. 112). The patterns identify several approaches that can be used by those developing modeling tools with flexibility requirements. They intend to represent the most relevant approaches in this area [CA13].

Patterns of Adaptive Object-Models. An Adaptive Object-Model (AOM) is an architectural pattern based upon a dynamic meta-modeling technique where the object model of the system is explicitly defined as data to be interpreted at run-time. The object model may encompass the full specification of domain objects, states, events, conditions, constraints and business rules. Several design patterns have before been documented and describe a set of good-practices within this domain. These patterns describe key concepts of object-oriented meta-architecture that are essential do AOMs

– Everything is a Thing, Closing the Roof, Bootstrapping and Lazy Semantics (p. 114) – as well as patterns specific to the evolution of data and metadata in the context of AOMs, which can be used to track, version, and evolve information at several abstraction levels – History of Operations, System Memento and Migration (p. 114) – and a pattern to reconcile adaptivity with the programming language's object model – Language Piggybacking (p. 191).

5.4 Patterns of Consistent Software Documentation

The artifacts created and evolved during the software development process are forms of captured knowledge. They are of different natures and capture several types of information. Some of them are more structured and formal, and thus specialized; others are more flexible and may be used to express virtually any intended topic.

Despite being useful for any software project, the value of software documentation depends on its ability to convey accurate information. It is therefore imperative to assure that it remains consistent. Software documentation often focuses on capturing informal, unstructured, human-oriented information. Consequently, ensuring its consistence is a hard to automate process, and therefore highly dependent upon human intervention.

Also, software systems evolve frequently, implying changes in code artifacts along with their related documentation (e.g. requirements, architecture and design documents). In fact, one of the highest costs of maintaining documentation for a large system is to ensure that it is kept in-sync within itself and among its related artifacts, a practice that may require continuous review.

In this context, inconsistencies essentially occur when particular information evolves independently, without the evolution of other related parts. Among other reasons, this may happen because: (a) the author lacks a global knowledge of all artifact dependencies; (b) a particular change cascades into multiple other changes, thus making harder the task of manually tracking them; or (c) as a deliberate way of reducing the maintenance effort.

It is important to note that documents with different subjects, target audiences, and frequencies of use are likely to require different degrees of accurateness. This means that deliberately allowing the documentation to become outdated may be a reasonable choice in some circumstances. For example, some types of documents are useful only within a specific time period, and there may be no value in updating them beyond it.

It is therefore advisable to take an agile approach towards documentation, producing and evolving it *just enough* and *when needed*, to answer the needs of the project at hands.

Patterns addressing the topic of software documentation have previously been documented. The book *"Agile Documentation: A Pattern Guide to Producing Lightweight Documents for Software Projects"* [Rö3] introduces a set of patterns covering a wide scope of concerns in the production of software documentation, and the pattern language *"Patterns for Documenting Frameworks"* [ADo5a, ADo6a, ADo6b, ADo6c, ADo7] has focused on framework documentation in particular. Ambler's work on agile documentation and modeling is also very relevant to this topic [Ambo2, Amb].

Although having some commonalities with the aforementioned works, the patterns presented in this section keep other important issues in view but address software documentation mainly from a consistency standpoint. They are meant to support teams on the selection of documentation-related tools, and to help tool developers to implement the most appropriate techniques to support documentation consistency.

5.4.1 Overview

An overview of the patterns, and how they relate to each other, is shown in Figure 5.1.

Since the same information may exist, partially, or totally, in more than one document, there are implicit relations between those contents. With no easy way of recovering these relations, the effort of maintaining consistency increases, as information may be duplicated and scattered over several documents. INFORMATION PROXIMITY (p. 66) focuses on establishing and using explicit relations among different artifacts.

Software artifacts change to better respond to new needs, and documentation is required to accompany this evolution. However, due to the aforementioned intrinsic relations between different parts of documentation, it is common that locally introduced

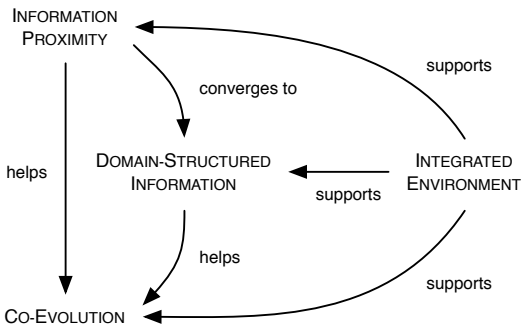


Figure 5.1: Pattern map of the consistent software documentation patterns.

changes may render other documentation parts inconsistent or obsolete. Co-Evolution (p. 71) focuses on strategies to update documentation while maintaining its consistency.

Domain-Structured Information (p. 74) deals with structuring contents, with the main objective of automating the process of assessing consistency.

Different types of artifacts may require different types of authoring tools. Integrated Environments (p. 76) articulate the use of different tools and allow them to be used uniformly.

5.4.2 Information Proximity

Software documentation can be captured as a set of documents of different types and purposes. Thus, they may sometimes address the same information from different perspectives. However, as documentation evolves, the effort of keeping them consistent rises due to the proliferation of duplicate and closely related contents.

Problem

How to preserve documentation consistency when fragments of related content are scattered across documents?

Having related information fragments in **close proximity** of each other, and having **multiple uses** of the same information, across a set of different documents, are common needs in software documentation.

There is value in a well established **separation of concerns** among different artifacts, as it allows them to be reused more easily, but there may exist the need to tailor them to specific contexts, as to maintain a **high fitness for purpose**.

Solution

Keep related information fragments easily accessible from each other, using a single source, links, views, or transclusion, for example, so that it may be easier to assess if they are in-sync.

The concrete approach to keep related information fragments close to each other greatly depends on the purpose of the document being produced. Four different alternatives are considered here and further described in the following sections.

The use of **links** is a good choice if the related contents are not meant to be part of the same document, and **single-source**, **transclusion** and **views** may be used otherwise. **Single-source** may be a good choice if the related contents can be made part of the same artifact (e.g. source code and API documentation are both frequently

expressed as text in the same file), and **transclusion** or **views** should be used when the related contents already exist in other artifacts. A **view** may be seen as a special case of **transclusion**, in which all contents already exist in other artifacts, while, with **transclusion**, the document being authored has contents of it's own, and only parts of it are used from other documents.

Variant: Links

Use explicit relations between different resources, so that related contents are kept separated and readers may easily travel between them.

Creating links between contents allows to explicitly relate them, while keeping them as separate entities from both the authors' and readers' viewpoint. Links allow readers to quickly reach related pieces of information, hence making easier the process of maintaining them consistent.

The following consequences should be considered when applying this technique:

Web of Documents. Links leads to the creation of relations among the existing contents, forming a web of related documents.

Reuse. Although not a reuse technique *per se*, links reduce the need to duplicate contents.

Reachability. Even if information is created, stored and presented separately, one may easily reach related contents. However, it is worth noting that the achieved proximity between contents is not necessarily bidirectional – linking the artifact A to artifact B makes B closer to A, but the opposite isn't necessarily true, e.g. if backlinks are not supported.

Effort. Removing duplicated contents requires an additional effort to detect common information fragments and to restructure them accordingly.

Variant: Single Source

Capture related information fragments in the same artifact, so that they may be easily maintained close to each other.

Although this isn't possible for all kinds of information, some can be captured together in the same artifact. Doing so allows related information fragments to be kept consistent since the author can more easily travel between them. This is thus a form of

physical information proximity as the contents are stored together with the goal of being presented together to readers and authors.

The following consequences should be considered when applying this technique:

Single artifact. The reader is presented with a document based in a single artifact, although including different types of information.

Reuse. Capturing different information fragments in the same artifact makes them less modular, and thus more difficult to reuse.

Reachability. Related information is stored and presented to the reader close to each other.

Effort. The flow of creating documentation is better than if these contents were kept separately.

Variant: Transclusion

Import the contents of an information fragment into a document by using a reference to it.

Isolating fragments of information as individual units eases their use for different purposes. Transclusion consists of creating references to information fragments on a document in such a way that they are presented to the reader as part of the document itself. Documents can be composed this way to fit the author's intent. Transclusion is a form of *virtual information proximity* since the contents are stored separately.

The following consequences should be considered when applying this technique:

Document-Oriented. Although leading to the creation of individual information fragments, the final result is a *document* tailored to a specific purpose.

Reuse. Abstracting information into individual units also allows them to be reused more effectively.

Reachability. Information that may me created and stored separately is presented to the reader near each other. However, this proximity may be unidirectional – transcluding the artifact A into artifact B makes A closer to B, but the opposite isn't necessarily true.

Effort. The flow of creating documentation may be hindered when authors are faced with the need to abstract existing information into new distinct units.

Variant: Views

Create a virtual document, composed by different individual fragments of information.

Views may be called *virtual documents*, as they have no content of their own. Instead, they filter, transform and combine contents according to a desired format into a single document. It is thus a form of *virtual information proximity* in the sense that contents are stored separately.

The following consequences should be considered when applying this technique:

Document-Oriented. The final result is a *document* tailored for a specific purpose, although contents may be woven together from several sources.

Reuse. Weaving contents into a view is an effective way of reusing them.

Reachability. Information that may be created and stored separately is presented to the reader near each other. However, this proximity may be unidirectional – the contents may be *close* to one another on the context of a given view, but it may not be possible to reach one from the other outside of this context.

Creating Heterogeneous Documents

Using artifacts of different types to create a document gives rise to a heterogeneous document. This may be achieved by using techniques, like **single-source**, **transclusion** or **views**, but different types of information may require different authoring tools, making information fragments more difficult to combine. From these techniques, **single-source** in particular is restrained to formats that may be combined into the same file, while **transclusion** and **views** may more easily be used with different types of contents.

Related Patterns

This pattern helps Co-Evolution, as keeping related contents near each other helps to change them together. Moreover, the creation of explicit relations frequently implies conferring more structure to the contents, which may converge to Domain-Structured Information.

As with the other documentation patterns in this chapter, Information Proximity greatly benefits from an appropriate tool support, which may be leveraged by an Integrated Environment.

WIKIS [Rö3] address the use of **links** but goes beyond the creation of explicit and navigable relations between resources, addressing the collaborative nature of this kind of systems.

Single-source is an approach similar to the one taken by the CODE-COMMENT PROXIMITY pattern [Rö3], but goes beyond source code and comments, not restricting itself to any particular type of information.

Transclusion is similar to the IMPORT BY REFERENCE pattern [Rö3], although it focuses on consistency maintenance.

Known Uses

Hypertext-based systems in general, of which wikis are a good example, allow to establish **links** between related resources. The term **transclusion** appeared initially in the context of hypertext-based systems. For example, Mediawiki, the wiki engine powering Wikipedia, uses this concept to allow the inclusion of repetitive blocks of content. XSDoc [ADP03, AD05b] is a wiki engine oriented for software development that uses **transclusion** to weave together heterogeneous artifacts, thus giving origin to heterogeneous documents.

Using the technique of *Code Annotations* (based on **single source**), documentation (or parts of it) can be generated from a unified representation of textual descriptions and source code. It is primarily used in the creation of API documentation and is supported by several tools: Javadoc [Fri95] is one of the first known uses of the technique, as is Autoduck [Artoo], a tool supporting code annotations in C++. The .NET framework uses XML in code annotations to produce compendiums of API documentation (CHM, HTML, etc.), in-editor assistance, and code-completion.

Views are frequently the product of an automatic generation process, in which several contents are combined according to a pre-established document form – some tools exist that support this approach [BM06].

Literate Programming (LP) [Knu84] combines textual descriptions and source code in a **single source file**, and provides the mechanisms to extract such different contents to different artifacts whenever required. The LP tool set *dotNoweb* [Sou05] further allows combining textual descriptions and source code with diagrams expressed using the *dot* language. LP systems also usually provide a form of **transclusion**, by allowing the creation of information fragments – *chunks* – which can then be (re)used multiple times across several documents.

Elucidative Programming [VN02] is a documentation technique that relies on

the creation of **links** between source code and documentation, allowing to mutually navigate between them.

Several office software suites, such as Microsoft Office and OpenOffice, allow combining different kinds of artifacts in a same document, also resulting in heterogeneous documents. Some uses of Literate Programming, such as VDMTools, directly parse and write *.rtf* documents, which have native support for images.

5.4.3 Co-Evolution

Software documentation can be captured as a set of text documents of different types and purposes. Thus, they may sometimes address the same information from different perspectives. However, as documentation evolves, the effort of keeping them consistent rises due to the proliferation of duplicate and closely related contents.

Problem

When to update a related piece of information in documentation?

Changes are made by the authors, who have the introduction of added value in view. However, changes required to ensure consistency don't always provide **immediate benefits**, and may shift the author's main **focus**.

Furthermore, the primary **goal** of the project will not always be the same. For example, during an inception phase, the **change rate** at which documented artifacts evolve is usually high. This means that changing just enough of the related information fragments might be the best choice. On the other hand, deployment phases may benefit from producing documentation with a higher level of **detail**.

Finally, **tracking** all the required changes may be difficult to carry out without any kind of **auxiliary support**, since it is easy to disregard global consequences during local modifications.

Solution

When a change is introduced, update the related information parts.

If all the related pieces aren't updated at the same time, they may grow harder to resync as time passes. Two variants to the co-evolution of contents are considered here and are further described on the following sections.

Synchronous co-evolution is a good option when it is important that documentation is kept consistent at all times, or if the effort of recovering consistency at a later

time is high. **Time-shifted co-evolution** may be used when the effort of recovering consistency is reasonable. This may happen when it is not difficult to assess the existence of relations between contents and the presence of inconsistencies between them.

Variant: Synchronous Co-Evolution

Whenever a change is introduced, update every related piece of information.

Although the quantity of information to be updated may be considerable, the most reliable way of ensuring consistency is to update all related information at the same time. Changes are made in small increments, in order to reduce the risk of forgetting to update something.

The following consequences should be considered when applying this technique:

State. Documentation is always in a consistent state.

Focus. The focus of the author on the task at hand is harder to maintain, as some of the changes she is required to do are not directly related with her main goal.

Effort. Introducing a change to a document carries a *higher up-front cost* – it may take more time than expected, as all the related contents will have to be updated at the same time.

Efficiency. If a particular fragment has several others that depend on it, and it has a high rate of change, it may be inefficient to keep consistency at all times.

Variant: Time-Shifted Co-Evolution

Whenever a change is introduced, provide mechanisms to track the pending related changes, and update the most relevant pieces of information only when needed.

Related contents don't need to be updated simultaneously if the changes that are made are in some way recorded. Authors will be able to, at a later time, assess which are the pending related changes, and evolve documentation to a consistent state as soon as they are addressed.

For example, using the concept of auditable document (see Section 5.4.3) authors may gain more awareness of the required modifications, facilitating the detection of changes that are still to be applied.

The following consequences should be considered when applying this technique:

State. Consistency is not kept at all time.

Focus. The author may focus solely on the task at hand, leaving related changes for later.

Effort. Only the changes that bring *short term benefits* are required to be made, and related changes may be deferred to a later time.

Efficiency. The task of updating documentation is distributed across the development process, as documentation may be updated only when necessary. However, the author may be faced with the additional effort of tracking which information needs to be updated, even if tools that support this task may exist.

Creating Auditable Documents

An auditable document makes it possible to assess at any time who, how, why, and what has been produced, by tracking information regarding the authoring process.

Being able to follow and understand how a document is evolved makes the entire process more **transparent** and **traceable**. However, it is important to note that the tracking mechanisms may increase the **complexity** of authoring the document, and the extra information that is recorded may increase the **storage space consumption**. Furthermore, for heterogeneous documents, tracking the evolution as a whole may involve tracking different types of artifacts.

Related Patterns

Domain-Structured Information supports Co-Evolution, since making richer information available allows tracking the information that needs to be co-evolved in greater detail. Information Proximity helps this pattern too, since having related contents easily reachable from one another assists in determining which contents are affected by a particular change.

Some patterns already describe the use of auditable documents in more concrete scenarios, namely Document History [Rö3] focuses on maintaining a list of past versions of a document, and Annotated Changes [Rö3] provides a way to directly record, inside a document, which of its parts have recently been modified.

Known Uses

Literate Programming and Code-Annotations, such as Javadoc, may be regarded as a way of supporting **synchronous co-evolution**, as providing INFORMATION PROXIMITY helps to co-evolve related information parts simultaneously.

Solutions that allow auditable documents to be produced support **time-shifted co-evolution**. Wiki engines and version control systems are good examples of such solutions, which allow to track how documents evolve and support assessing which changes are required to maintain consistency.

It is common for text processors to provide a *track changes* feature, which is a form of ANNOTATED CHANGES. This feature may be used by authors and readers to track the changes the document has recently gone through. Although this makes the document auditable to a certain point, it is usually very limited in time.

5.4.4 DOMAIN-STRUCTURED INFORMATION

Free-text documents are often an important fraction of a software project's documentation. They follow a text-oriented structure, using elements such as titles, paragraphs, lists, tables, etc. Although these elements allow a lot of flexibility, the degree to which a free-text document is useful depends on how well it accurately expresses the intended ideas. Moreover, the same piece of information may be better conveyed using different perspectives, intrinsically related to each other.

The main reason why maintaining documentation requires continuous review is that relations between documentation parts aren't explicitly formalized. This decreases the capability to automatically process it, i.e. in order to automatically assess its consistency.

Problem

How to structure the information in documentation?

As mentioned before, textual documentation is a **flexible** way of capturing knowledge. While this flexibility is an important asset, **formalizing** the content itself makes information less subject to multiple interpretations, and allows it to be automatically processed.

However, the mechanisms used to allow a degree of formalization higher than that provided by simple textual descriptions may affect the **simplicity** in producing documentation.

Solution

Organize contents according to their domain, so that the information form directly relates to domain concepts.

Textual documentation doesn't provide the mechanisms to formally express the relations between the concepts being documented. Structuring the contents around the domain concepts provides the support to automatically assess the existence of inconsistencies, and prevents the introduction of new ones.

The following consequences should be considered when applying this pattern:

Flexibility. Some flexibility is lost whenever information has to follow a predefined structure.

Automation. The use of a domain-oriented structure with well defined semantics makes information less open to different interpretations, and allows it to be processed by computers.

Related Patterns

The individual information units often required by Information Proximity tend to converge to Domain-Structured Information, as the advantages of organizing the contents around domain concepts emerge. This pattern also supports Co-Evolution, as it provides a richer base of traceable information.

As with the other documentation patterns in this chapter, Domain-Structured Information requires appropriate tool support, and may benefit from the use of an Integrated Environment.

This pattern is similar to Structured Information [Rö3], in that it also addresses how documents' contents are organized. However, Domain-Structured Information focuses on formalizing contents according to the information's domain, with the aim of automating consistency assessment, while Structured Information focuses mainly in structuring contents to ease the perception of the readers.

Known Uses

Code comments are a form of source code documentation. Code annotations, such as Javadoc comments [Fri95], add an additional level of structure to source code comments, formalizing information elements with a lower granularity. Javadoc allows describing elements such as method parameters, authors, creation dates and references, among others.

Semantic Wikis support Domain-Structured Information, and some semantic wiki engines may automatically detect existing inconsistencies with the use of reasoners [DRR+05].

Some wiki engines allow templates to be applied for very specific purposes. Mediawiki allows the creation of sidebar templates, through which one may provide structured information.

Systems taking an object-oriented approach to documentation have also been use in the past [Sam94, CS96].

5.4.5 Integrated Environment

Working with different kinds of artifacts frequently implies the use of specialized and independent tools for each of them. Although such artifacts are sometimes strongly related, these tools don't necessarily interoperate, making the artifacts more difficult to combine and confront, and the authoring environment heterogeneous and more difficult to use.

Problem

How to support the maintenance of consistency between independent artifacts with related content?

Tools that deal with a **wide range of artifacts** usually provide a more **homogeneous** and **interoperable** environment, although they tend to be not as **powerful** and **simple** as **specialized tools**.

Solution

Use an integrated environment, where several types of artifacts and their relations may be maintained uniformly.

An integrated environment goes beyond the capabilities that general purpose tools possess. It supports handling several types of artifacts, providing specialized features for each of them and an infrastructure through which they interoperate.

This supports strategies of documentation maintenance that focus on bridging related information parts regardless of their nature.

The following consequences should be considered when applying this pattern:

Specialization. Integrated environments strike a balance between a generic approach, in which tools may handle several types of artifacts with a basic level of

functionality, and a specialized approach, in which exists a deeper support for a selected set of artifact types.

Simplicity. While potentially making each tool more complex individually, their overall simplicity is increased by providing a more homogeneous usage.

Interoperability. An integrated environment coordinates the several tools it provides, and supports their interoperability.

Related Patterns

INTEGRATED ENVIRONMENT directly contributes to the remaining documentation patterns of this chapter by orchestrating the several tools involved. It is also directly related to the pattern FEW TOOLS [Rö3], which addresses the notion that supporting the creation of documentation with too many and unconnected tools may become a burden to authors.

Known Uses

Eclipse and Visual Studio are examples of integrated environments that combine different kinds of artifacts and tools, supporting and articulating their work.

Trac [Edga] and Redmine [Lan] are Web-based environments that integrate different kinds of information, including textual descriptions supported by a wiki, source code browsing, milestone management, issue-tracking, etc.

5.5 Patterns of Information Classification

In the context of knowledge work, it is expected that as the available information grows, one would be more effective in his tasks. Unfortunately this is not always the case, and the value of information frequently decreases as the quantity of information increases. This apparent contradiction is due to our human limitations in processing high quantities of raw information.

This section looks into six patterns for classifying and improving the access to information. Some of these solutions have been used since the 4^{th} century [Wel94], and are nowadays very well known in the domain of information science. Others came into being on the context of the Web, even though they conceptually share a lot with "older" solutions, but all are used as means for information seeking and retrieval. In one way or another, they can all nowadays be seen pervasively in software systems.

The main audience for these patterns are those wanting to make information quickly reachable, in the context of software systems. Depending on the kind of system, they can be either developers or users of the system. Although the patterns don't lead to a specific implementation, their implications easily crosscut the design of a system, from how the data modeling is done, to how information is perceived and interacted with through the user-interface.

To a lesser extent, we believe these patterns may also be useful to those wanting to take their first steps into information indexing, and need to gain a better understanding of the different concepts involved.

5.5.1 Overview

These patterns were mined from the experience gathered by the authors while developing software systems – some of them in the information science domain – that use these techniques to make information accessible.

Two approaches to accessing contents – searching and browsing – have proven useful in different contexts. While search provides immediate results, browsing allows an exploratory approach to finding contents, which is key when information needs are ill-defined. The patterns described in this section focus mainly on supporting the access to contents through browsing.

Organizing, Classifying, Indexing

These three concepts are used throughout the patterns, but the differences between them can sometimes be subtle. They can work together to support the same overall goal – to ease the understanding and access to contents. To **organize** is to provide an *order*, that is, to systematize the way in which the contents are recorded and conveyed, so that they can be more easily understood. On the other hand, to **classify** is to assign the contents to *classes*, that is, to group them according to common features – it implies abstraction, and a specific kind of organization. At last, to **index** is to provide the key topics or the classes of the contents as access points to those contents; the emphasis is on how readers can use those common features to actually find and delve into the contents.

The patterns below address these three concerns to some degree. They are Information *Classification* Patterns because they focus mainly on how the different topics of the contents are abstracted and represented.

The Patterns

The first pattern of this set is the INDEX. Indexes can be elaborate structures, but, in their simplest form, they are lists of terms, usually organized alphabetically. In the context of publishing, the word *index* specifically denotes an alphabetically ordered INDEX of subjects, usually appearing in the back of the document, but unless noted otherwise, the term *index* will here always be used in the most general sense, as will become clear in the description of the pattern.

The creation of an index requires the use of a representation language, which supports expressing its entries. When used to represent the information of an index, these languages are called *indexing languages*. The most expressive ones are used for other purposes too, as they are able of representing knowledge in general. Four of the patterns – TAXONOMY, THESAURUS, ONTOLOGY and FOLKSONOMY – are about such languages. Directly or indirectly, they support the creation of INDEXES, and they all strike different balances between the effort of creation, the effectiveness of knowledge capture, and the ease of use. At last, the CONTROLLED VOCABULARY pattern describes a general approach to disambiguating the meaning of terms; it is key in THESAURI and often used with ONTOLOGIES.

Figure 5.2 depicts the relations that were just described, and will be explored in greater detail in the description of each pattern.

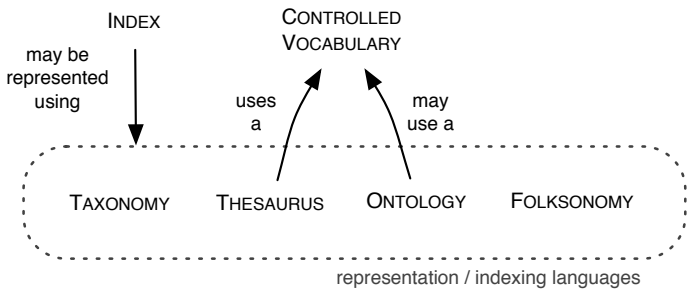


Figure 5.2: Pattern map of the information classification patterns.

INDEX (p. 82) – Supports readers in finding the contents they seek more efficiently;

TAXONOMY (p. 85) – Allows representation of information along an hierarchical structure with loosely defined semantics;

THESAURUS (p. 88) – Provides more semantics and expressiveness when representing information, allowing related subjects to be connected;

ONTOLOGY (p. 91) – Provides even richer semantics and expressiveness, and allows representing information as a graph of related subjects, connected through arbitrary types of relations;

FOLKSONOMY (p. 94) – Supports a collaborative approach to classifying information, but not without a loss in semantics and expressiveness.

CONTROLLED VOCABULARY (p. 97) – Allows referring precisely to subjects, by disambiguating the meaning of terms.

How the INDEX is represented determines how readers will be able to use it. Its *coordination* is one of the factors to consider during the index creation. During this process, several concepts may be combined to create the index entries, in which case it is called *Pre-coordinate Indexing*. With the opposite approach, *Post-coordinate Indexing*, index entries are based on elemental concepts, which are only pulled together during the access phase, by the reader [Teno8].

Post-coordinated indexes are better to explore the several dimensions of the contents, because they offer no restriction as to which terms to combine, but they don't hint the reader as to which combination of terms might be more interesting to explore. It's also worth noting that, when indexes need to be represented in a static form (e.g., printed on paper), the advantages of Post-coordination are lost, as there is no easy way to obtain the set of contents indexed by more than one term. Pre-coordinate indexes are usually used in such cases [Lano3, p. 50].

5.5.2 General Forces

These patterns share a set of forces, which are depicted by Figure 5.3. These general forces influence the use of indexing languages and the creation of indexes. They affect almost all of the patterns, and they do so in different ways, as the application context and goal of each pattern also varies. Each of the patterns will describe in more detail the aspects of these forces that matter most in each case.

Semantic Richness. How much information does the index contain? Or, more specifically, how unambiguous and meaningful are the index entries? A semantically rich index will be better equipped to guide the reader, although it requires more effort to create. This is especially true if it's created by a group of individuals, as it requires agreement to be reached about the meaning that each entry conveys.

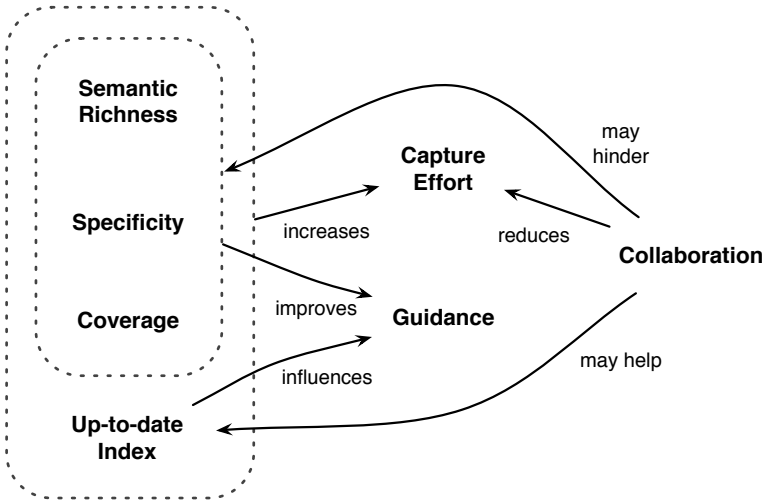


Figure 5.3: Relations between common forces of the information classification patterns.

Specificity. Index entries that reflect the same level of specificity of the contents will better align the needs of readers to the contents that they need to be guided to. Namely, entries that are more general should lead to general contents, and entries that are more specific should lead to specific contents.

Coverage. The extent to which the index entries cover the entire contents. By ensuring that the most representative subjects of the contents are part of the index, the readers will be better guided, helping them not to miss useful bits of information.

Up-to-date Index. An index that reflects the current contents makes them easier to find. On the other hand, the reader may never get proficient using the index if it is constantly being completed and updated. Keeping the index of a large and evolving body of information up to date requires a lot of effort, and is usually impracticable unless it's done collaboratively.

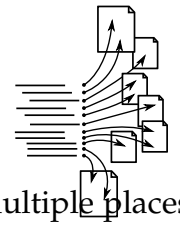
Capture Effort. Creating an index may be a costly process. In part, that is due to the time that it takes, but it is also because capturing that knowledge, representing it with elaborate index structures, requires uncommon analysis and abstraction skills. Making it a collaborative process may reduce this effort.

Guidance. An index should be useful for multiple audiences, with different degrees of knowledge on the domain at hand. Readers that are not knowledgeable on a area, and have fuzzier needs, should still be able to use the INDEX effectively. Some knowledge is always needed to carry out the educated guesswork that

allows getting to the *right* index entry rapidly. Just a little knowledge of the domain may be enough for a reader to recognize the index entry he needs when confronted with it, although he has to go through the whole index to search for the subject, which can be very time consuming, or even impracticable. But outsiders to the area may not be able to even recognize the entry that would lead them to the desired contents. Building quality indexes, with more and better contextual information around them (e.g., more Semantic Richness, Specificity, Coverage, and ensuring they are Up-to-date) provides more guidance to newcomers to the area, even though it requires more effort from their creators and maintainers.

Collaboration. Obtaining the collaboration of authors and readers of a body of information can be an effective and inexpensive way to create its index and keep it up to date. However, this may happen in expense of quality, as not all participants will possess the skills or be willing to devote a lot of time to the task.

5.5.3 INDEX



Consider that there is a body of information, which you would like to make available. Due to its extent, one can't just scan it quickly to find what she is looking for. Some topics may appear in multiple places in the contents. And also, after you find a particular piece of content, you might need to come back to it again later.

Example

Suppose you buy a subscription for an online book, about the very newest and exciting programming language that everyone is now using. Imagine for a moment that the author provided only a list of pages, and the book has no table of contents, nor a traditional back-of-the-book index. You have a considerable amount of pages ahead, but you already know something about the language, so you would like to skip the first sections, and go directly to the first hands-on exercise. You would also like to know where to find help on the language's class library, so you can reference to it as you do the exercises.

Problem

Without some sort of a guide, and given a non-trivial amount of contents, the effort of trying to go through it all in order to find a particular piece of information may be overwhelming.

As readers, we want to **quickly** find the contents we need. We can go through a text document exhaustively searching for what is relevant for us, but again we don't want to take too much time doing it. We may choose to just skim through the whole document instead, but we might miss something relevant if we do so. Moreover, when going through the body of information we have to **repeatedly** assess if each piece of content is relevant or not.

Both **reading** and **evolving** the contents should be easy to carry out, but the most elaborate indices are usually more difficult to keep up to date.

It's important to lower the barrier to **reading** the contents, although that might sometimes happens in expense of how easily they can be **evolved**. Both tasks should be easy to carry out.

We want the index to have a good **breath** of coverage of the contents (scope) and a high **specificity** of its terms (but no more than the contents convey), as this better aligns the contents to the needs of readers. These goals should be balanced with the high **cost** that creating a quality index entails [Lano3, p. 28–29].

Solution

Analyze your information to identify the most important subjects, represent them and organize them systematically, making each of them refer back to the contents that they respectively describe.

In this process, make sure to include all topics that are treated in the contents and that may be of interest to the readers. Represent such topics in the index, using terms with the same level of specificity of the contents [Lano3, p. 36]. How the subjects are actually represented in the index, and the amount of information that the index contains, should depend on how the readers will want to use it.

Instead of going through the whole contents, the information seeker goes through the index *entries*. Each entry has a *locator*, which refers back to the place(s) where the reader may find that subject in the contents. An index may thus be said to be a simplified view – an abstraction – of the subjects that may matter to the reader. Indexes are crafted to allow readers to search through the subjects without having to deal with the whole contents.

By reducing the search universe to a smaller set of terms, we are improving the search **efficiency**, maintaining an equivalent **effectiveness**, because these terms were the result of an analyses process, which selected only the most important topics.

By making the index vocabulary reflect the contents, information seekers will be guided to contents with the same level of **specificity** they have searched for.

Example Resolved

A Table of Contents provides an overview of its book, but most importantly, it provides an index for the book's sections. So, it supports the reader in finding a particular section, and reaching it easily. Going back to the example presented in the beginning of this pattern, the reader would use the Table of Contents to find the section for the first hands-on exercise, and follow the given page number reference. A traditional back-of-the-book index, which is usually alphabetical, could also help in finding the description of some of the class library functions used in the exercises.

Both – tables of contents and a back-of-the-book indexes – are good examples of INDEXES. They don't have to be used together like in this example, but are often synergistic.

Known Uses

A menu bar, in a window-based software system, can be considered an index to some extent. It is an abstraction of the system's functional parts that organizes the way users have access to them.

The TeX typesetting system supports the creation of indices, such as a table of contents, list of figures, list of tables, etc. The author annotates the contents, and the system will automatically create a list of entries, together with the locators to the respective contents. Printed books often use one, or both, of the INDEXES mentioned in the example sections above. While a Table of Contents of a book indexes its constituent parts (chapters, sections, etc.), a traditional back-of-the-book index indexes the book's contents by subject.

Wikipedia includes several index pages. The *list of wiki software*[2], for example, is a page listing and linking to pages about wiki software packages.

Related Patterns

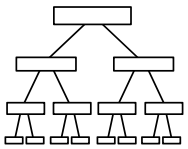
The choice of an INDEX representation language depends on the context at hand. If the contents are not updated very frequently, maybe because they are in print, TAXONOMIES and THESAURI are good options – they assume a closed domain, and

[2] Available at http://en.wikipedia.org/wiki/List_of_wiki_software.

that the information of the index is represented before indexing is actually done (i.e., before the creation of the index *locators*). Choosing between the two depends on the complexity of the domain, and on how familiar information seekers are with the topics they will be searching for. The guidance provided by a TAXONOMY may be quicker if the reader already knows the domain area. Newcomers may also prefer a TAXONOMY if the contents are simple enough and the index entries can capture all of the most important pre-coordinations of terms.

The creation of ONTOLOGIES is often motivated by the need to capture and share information with rich semantics, while FOLKSONOMIES, are normally used when the need for collaboration is key. But they are both used to index contents that change frequently. They have appeared and become popular in the context of the Web, possibly because they are more practicable when tools that support such change are easily available.

5.5.4 TAXONOMY



Consider that you want to classify and create an INDEX for a body of information, to support the readers in quickly reaching the contents they need. Readers may have some prior knowledge on the subject they seek, and they can use it to search for the right contents. The frequency at which the contents are updated is not high, when compared with how many times they will be used.

Example

For some time, Amy has been keeping files in the "Documents" directory of her computer. She has already dozens of documents, a great many of them created by herself. She now needs to find a work-related document – a project proposal she sent to Mr. Smith during the past year.

She doesn't know the exact name of the file, so she tries narrowing her search. She orders files by date, but that still leaves her with too many files to go through. She also tries a text search, but that retrieves all the invoices she kept from her landlord – John Smith – as well as dozens of letters from one of her suppliers at work – Smith & Co.

Problem

Without knowing the exact term, it can take too much time to search through the whole index.

To sort out the index entries they seek, readers need additional information, which describes and contextualizes the entries, and that they can use to partition them. Although a **semantically richer** index implies a greater **effort** from indexers, it better **guides** the readers in finding contents.

Solution

Organize the index entries hierarchically. The meaning of the relations between parent and child entries may vary, so the index can be partitioned by different dimensions into several subareas.

Choose entries that cover the whole domain , and try to keep the taxonomy tree balanced. Add entries that can be easily understood by the readers when taken in context with the upper taxonomic levels. When creating the entries of a taxonomy don't try to make them stand on their own. To assemble meaning from an entry, readers will consider its context in the taxonomy. The same terms may convey different meanings when they appear in different points of the taxonomy.

Each taxonomic level relates to the upper level according to one of its dimensions. In case the same term is placed in more than one point of the taxonomy it does not mean that the same subject is being classified in multiple ways, but rather that different subjects are being represented. The order in which such dimensions are represented as parent-child entries should reflect the knowledge that we foresee readers may have, and the way they will seek the entries in the taxonomy.

From an indexing point of view, a TAXONOMY can be said to be a *fixed-vocabulary* language, as a pre-established representation of terms is taken as a basis for the indexing process. Taxonomic indexes are *pre-coordinated*, because each entry is a combination of terms that describe a group of other entries.

The partitioning of the index along a tree structure makes navigating it more **efficient**, as readers are able to eliminate from their search several index entries at once, when they belong to a subarea that does not interest them.

Also, the more a reader knows about a given domain, the more **efficient** she is navigating through a taxonomy of that domain – she will be quicker in grasping which dimensions the taxonomy is using to partition the index, and how she should navigate it to reach the intended index entries. Newcomers may have to explore the index first, before being able to use it efficiently.

But taxonomies are not without liabilities, and the level of **expressiveness** that they allow is one of them. In practice, it may be hard to group contents according to a single sequence of dimensions. Although you can try to anticipate which features of

each piece of information will be the most relevant to future readers, different readers may easily have very different needs.

The added **effort** of pre-coordinating and grouping together related terms makes a TAXONOMY harder to create and maintain when compared to using a plain list of terms. Only considering a high number of readers, and a low rate of updates does such effort pay itself easily.

Example Resolved

Realizing that she must stop going through all the files every time she needs one of them, Amy started organizing the files into subdirectories. She created three subdirectories inside the "Documents" directory: "Family", "Work" and "Friends". Inside the "Work" subdirectory she created some more subdirectories, one for each customer.

Whenever she needs to reach the project proposal for Mr. Smith again, she will rely on the directory structure to guide her. She will first open the "Work" subdirectory, then the "Mr. Smith" subdirectory, and finally go through the files there.

The top-most directories partition the larger groups of files, according to Amy's social groups (family, work, friends). In turn, the directories inside the "Work" directory are grouping the files by person – sender or receiver. Each level uses the dimension that better helps Amy navigating that particular group of files.

Known Uses

Several software systems use the concept of "folders" – container of items, used to organize them. Such items can themselves be other folders, and thus contain other items, forming a tree-like structure that can be regarded as a TAXONOMY. An example of such use is the Alfresco Content Management System[3], which has the concept of "Spaces". Alfresco Spaces are generic containers that behave much like "folders".

A Web Directory, like the Open Directory Project[4], can be seen as a TAXONOMY that classifies websites on the World Wide Web. Web Directories were once important to find Web resources, but due to their very high rate of change and growth, very few have survived in favor of full text search engines.

A Table of Contents, either digital or in print, may be seen as a TAXONOMY that organizes contents according to sections and chapters.

[3] Available at http://www.alfresco.com/.
[4] Available at http://www.dmoz.org/.

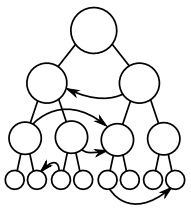
The Dewey Decimal Classification System is one of the TAXONOMIES with the most widespread use. Its goal is to cover all areas of knowledge, supporting the classification of books and other library items, and providing a way to easily find them, on the online catalog or shelves of a library.

Related Patterns

TAXONOMIES, like THESAURI, ONTOLOGIES and FOLKSONOMIES, can be used to represent an INDEX. Like THESAURI, TAXONOMIES assume a closed domain.

When information seekers are not newcomers to the domain area, or if the contents are simple enough, TAXONOMIES allow to reach information quicker, but otherwise, semantically richer indexing languages will provide better guidance.

5.5.5 THESAURUS



Consider that you want to classify and create an INDEX for a body of information, to support the readers in quickly reaching the contents they need. Readers may have **very different backgrounds and different levels of prior knowledge** on the subject they seek. The frequency at which the contents are updated is not high, when compared with how many times they will be used.

Example

Suppose a library keeps a set of documents about art, and allows readers to find them through an online catalog. John would like to find some documents about a famous painter, whose name he doesn't recall right now. All he remembers is that the painter was contemporary to Monet, and painted using the same style. Paul happens to also be looking for documents about the same painter, but all he knows is that he created the painting *"Dance at Le Moulin de la Galette"*.

Both John and Paul need to do some research before getting to the documents they need using the library's index. John will start by finding documents about Monet. He then uses these documents to learn that Monet was an impressionist, and then find documents about the painters of that movement, to finally recognize Renoir as the name he was missing. Searching by Renoir on the library's online catalog will finally reveal the documents he needs. Paul, on the other hand, will first need to search for

the name of the painting he knows, to find that it was painted by Renoir. He can then use the library's online catalog effectively.

Problem

Different people seeking the same contents need the index to guide them in different dimensions.

The same contents may need to be accessed differently, depending on the knowledge of the reader. To sort out the index entries they seek, readers need information that describes and contextualizes those entries. Although a **semantically richer** index implies a greater **effort** from indexers, it better guides the readers in finding contents.

Solution

Organize the index entries as a network of subjects. Define the meaning of each subject carefully, by using a Controlled Vocabulary*, and connect them with other, related, subjects.* More specifically, organize subjects according to five different elements/connections:

Broader/Narrower – Thesauri, like taxonomies, are organized hierarchically, but the semantics of such relations is better established than with taxonomies. *Parent* subjects are said to be *broader*, and child subjects *narrower*, in the sense that the scope of each child subject is narrower than the scope of the parent.

Scope Note – Each subject represents not merely a term, but a concept that is part of a Controlled Vocabulary. The meaning of the concept is defined through a *scope note*.

Synonyms – Other terms that may describe the same subject. Synonyms are *unauthorized forms* of the Controlled Vocabulary used to support the Thesaurus.

Topmost – Each subject has at least one topmost subject, which is the one that would be found by following the *broader* relations until the broadest possible subjects are reached.

Related – Refers to related subjects, which are not broader or narrower.

Despite Thesaurus' entries representing concepts, the terms are usually emphasized more than the underlying concepts, and Thesaurus are very often perceived as just a set of connected words. A Thesaurus allows a richer description of subjects when compared with a Taxonomy, as it supports expressing broader/narrower relations,

which have more concrete semantics than the hierarchical relations of a TAXONOMY. *Related* connections support expressing other (unspecific) kinds of relations. In spite of the added expressiveness when compared with TAXONOMIES, THESAURI are sometimes extended with even further attributes and kinds of relations.

To create a THESAURUS, identify subjects (i.e., index entries) at different degrees of abstraction, reflecting the different levels that may be found in the contents, from the very coarse-grained (general) subjects to the very fine-grained (specific) ones.

Represent in the THESAURUS all the knowledge in the field, and reuse is as often as needed. You should index the same piece of contents with multiple index entries, providing multiple access points, possibly to be combined, when searching the index.

THESAURI provide semantically richer connections between subjects, which makes them **quicker** to navigate for those without much prior knowledge on the domain.

THESAURI are meant to comprehensively cover their target domain, and are reviewed and updated only sporadically. They are, for this last reason, called a *fixed vocabulary*. This supports their **usability**, as it makes it easier for readers to learn how the thesauri that they use are organized.

Expressiveness is better than with a TAXONOMY, but THESAURI demand more attention to semantics, which can imply more **effort** during creation and maintenance. However, in practice, maintenance may not actually be harder than with a TAXONOMY, because the meaning of each THESAURUS' entry is more fine grained and better defined – it's easier to improve an entry while being confident that the rest of the THESAURUS remains consistent.

Example Resolved

Suppose that the catalog software, of the library of the example above, allows thesaurus-based indexing. The librarians have decided to built a thesaurus in the art domain, and are using it to index the documents that they are curating.

John will start looking for the elements he knows. He will first seek for the term "monet". He finds the corresponding subject in the thesaurus that he confirms to be the one he is looking for, upon reading the scope note, and by observing that it is *narrower* term of the "Painter" entry. He quickly goes through that thesaurus entry, and finds out that "impressionism" is a *related* thesaurus entry. He then looks at the remaining entries related with "impressionism", and recognizes "Renoir"as the painter he was seeking. He now just needs to follow the index locator(s) for that entry, to reach the documents about Renoir.

Paul, on the other hand, will start seeking for the term "Dance at Le Moulin de la Galette". He finds it, and too sees "Renoir" as a related entry. All he has to do now is follow the locator(s) to reach the documents he needs.

Known Uses

GISA is a software product for creating records and descriptions of archival documents, which uses a thesaurus-based index. Its Web frontend can be found in the websites of several Portuguese archives, like the Archives of the City Hall of Gaia[5] and the Archives of the University of Porto[6], among others.

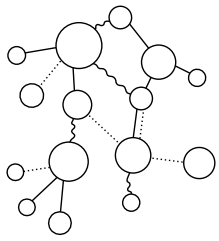
The *Index New Zealand Thesaurus*[7] was created to describe publications about New Zealand and the South Pacific in the areas of social sciences and humanities. It provides access to journal and newspaper articles.

Related Patterns

THESAURI can be used to represent an INDEX, like TAXONOMIES, ONTOLOGIES and FOLKSONOMIES. Like TAXONOMIES, THESAURI assume a closed domain.

THESAURI are better at guiding information seekers than TAXONOMIES. The entries of a THESAURUS form a CONTROLLED VOCABULARY in the sense that their meaning is established unambiguously.

5.5.6 ONTOLOGY



Consider that you want to classify and create an INDEX for a body of information to support the readers in quickly reaching the contents they need. Readers may have very different backgrounds and different levels of prior knowledge on the subject they seek. **Contents may frequently be created and updated, and they are likely accessed through platforms that enable collaboration, such as the Web.**

Example

A research institute has, over time, produced a large body of information. Some of these contents were recorded by the research groups themselves, in different software

[5] Available at http://arquivo.cm-gaia.pt/.
[6] Available at http://gisa.up.pt/pesquisa/.
[7] Available at http://innz.natlib.govt.nz/content/thesaurus/.

systems that they maintain. These contents are organized in different ways, depending on the system they were captured in; they have varying levels of structure, from information systems, to free-text documents, to raw experimental data; and they keep evolving as more results are found and documented.

Linda is researching on the area of automated software testing, and would like to know which results her colleagues have achieved in this area in the last few months. She hopes to find work to build upon, or find researchers of other groups to collaborate with.

The institute provides a list of systems that it uses to capture contents internally. Each group works on a specific subarea, and knowing this could help Linda find the systems with the contents she needs. However, there aren't groups working specifically on automated testing, and almost all of the groups have done some automated testing at some point. If she wants to make sure to find all the contents she needs, Linda will have to search through all the systems and their information.

Problem

Without a rich and accurate representation of the contents, an index is not able to guide a reader effectively.

To sort out the index entries they seek, readers need additional information, which describes and contextualizes such entries. Although a **semantically richer** index implies a greater **effort** from indexers, it better **guides** the readers in finding contents.

Solution

Organize the index entries as a network of subjects. Define the meaning of each subject, and connect it with other, related, subjects. More specifically, organize subjects according to elements such as individuals, classes, attributes *and* relations, *among others.*

An Ontology is a *formal, explicit specification of a shared conceptualization* [Gru93]. It may gather a collective understanding on a given area and be open to, and constantly updated by, a group of people. Several ontology languages exist, but common components include *Classes*, *Attributes*, *Relationships* and *Individuals*. Subjects may be defined by classes or individuals, and are characterized by attributes and relationships. Attributes and Relationships are themselves described by classes, and this mechanism allows the language to be extended as needed. Given its formal nature, an Ontology is very fit to use a Controlled Vocabulary.

The ability of expressing virtually any kind of attribute and relation makes ONTOLOGIES able to provide a **richer description** of subjects when compared with a TAXONOMY a THESAURUS or a FOLKSONOMY, and has thus the capacity of **guiding** information seekers more effectively.

Even though an ONTOLOGY may be **open** and constantly updated by a community, it may sometimes prove to be a difficult endeavor to reach a **consensus** over the conceptualizations.

Example Resolved

Going back to the example presented in the beginning of this pattern, the institute can build an aggregator that gathers contents from each system, and provides a unified and abstracted representation of them. This ontology can be used as a global index that users use to reach the actual contents. Although the source contents need to be semantically rich to be aggregated, the ontology can be completed with additional information to make other contents accessible through the index too.

Linda now uses the ontology-based index to find an entry about automated testing, and follows the index locators to free-text documents maintained in a system used by the Artificial Intelligence Group, and to some unit-test coverage data that was used for creating software visualizations by the Computer Graphics Group. Linda is directed to the right systems and, more specifically, to the right contents within the system.

Known Uses

Semantic MediaWiki[8] is an extension to the MediaWiki wiki engine. It adds the ability to annotate the contents of a page, conferring it semantics. This data can then be queried, by one or several of its dimensions, and the results provided, from within a wiki page, as an access to the contents in question. Among other features, it supports exporting data as OWL[9], an ONTOLOGY representation language based on XML.

In research, we can find several approaches to indexing contents with an ONTOLOGY. An example, among several others, is the work by Luances et al, which uses an Ontology to improve the query capabilities to a Geographic Information System [LPPS08].

[8] Available from http://semantic-mediawiki.org/.

[9] The full specification of the *Web Ontology Language* (OWL) may be found at http://www.w3.org/TR/owl-features/.

Plone Ontology[10] is an add-on for the Plone Content Management System that allows to collaboratively create an ontology that can be navigated and used to access the system's contents.

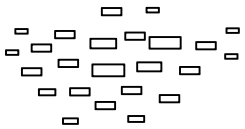
Related Patterns

ONTOLOGIES can be used to represent an INDEX, like TAXONOMIES, THESAURI and FOLKSONOMIES. Like FOLKSONOMIES, ONTOLOGIES assume an open domain.

If a rich description of contents is more important than supporting collaboration, an ONTOLOGY makes a better indexing language than a FOLKSONOMY.

The meaning of the elements of an ONTOLOGY is established unambiguously, and in that sense, it may be very close to using a CONTROLLED VOCABULARY. However, that's often not the case, as the elements of an ONTOLOGY don't necessarily have an authorized form.

5.5.7 FOLKSONOMY



Consider that you want to classify and create an INDEX for a body of information, to support the readers in quickly reaching the contents they need. Readers may have very different backgrounds and different levels of prior knowledge on the subject they seek. **The amount of contents is overwhelming;** they may be frequently created and updated, and are likely accessed through platforms that enable collaboration, such as the Web.

Example

Suppose you have created a software platform for amateur photographers through which they can publish their best works on the Web. You want to let users easily seek the contents they need, but you don't have the resources to hire a team to manually classify such a large set of pictures, and you also don't want to demand from users a lot of effort to classify their photos by topic.

Kate, an early adopter of the system, has just uploaded her photo album from the last five years, and she is now wondering how to find the pictures of the weekend she spent last year in Portugal. She would also enjoy knowing what other pictures of Portugal there are on the system, as she would like to find new places for her next visit.

[10] Available from http://plone.org/products/ploneontology

Problem

Indexing and classifying a large body of information is unfeasible or at least very costly to carry out.

Assuming the information is already recorded, one could consider assigning a team with the task of creating an index to ease all subsequent accesses. However, the **effort** of such an endeavor is usually very high. This is aggravated if such information is in constant **change**, in which case, the classification efforts cannot be limited in time, and have to follow the entire lifecycle of the information.

Those that are most **knowledgeable** about some specific contents are not external indexers but its own creators, as they are more aware of its context and domain. On the other hand, information creators are not necessarily aware of what makes a good index, and may lack the necessary **analysis and abstraction skills** to represent elaborate index structures.

Solution

Ask the creators or users of the information to identify the set of words that most accurately describe the contents, and tag *them with those terms.*

Those words are the *entries* of the index. By letting – and encouraging – users to assign descriptive terms to pieces of information, an index will emerge. It will not be defined up-front, but rather will gradually appear from the practice of collaboratively tagging contents [Fur10].

There is not a single way to seek contents using a folksonomy. Tags can be made available to information seekers as simple alphabetically ordered lists, or as *tag clouds*. Tag clouds present the several tags by laying them out in different locations, and using different font sizes and colors to highlight the relative importance of each one, usually directly reflecting the number of times they were used to tag some content (i.e., the number of underlying *locators*).

The final result is a *word index*, as opposed to a *subject index*. This kind of index does not make use of a Controlled Vocabulary, and thus its entries lack a **strong semantics**, leaving to the reader the job of figuring out if the contents tagged with a given entry actually refer to what he is looking for. However, this is also one of the biggest strengths of this solution; by reducing to a minimum the **effort** required in the analysis phase, the creation of the index is easy enough to be done by any information creator, in a **distributed** way.

Example Resolved

Going back to the example presented in the beginning of this pattern, the developers have just added to the system a feature that allows users to tag their own contents. Kate can now tag her photos with any term that she finds descriptive.

She adds the "portugal" tag to the photos that she took in Portugal. It happens that other users are using that tag too, so it doesn't take too long for Kate to be able to find other photos of this country. She will just follow the link on the "portugal" tag of one of her photos, and be lead to a full list of photos tagged with this term.

Known Uses

Folksonomies are very popular on the Web, and used by several successful websites.

Flickr[11] is an image and video hosting service in which users can classify their photos with tags. The community can then search photos by their tags, or they can seek photos through a tagcloud. Delicious[12] is another well-known service on the Web that makes extensive use of tags as a way to bookmark and describe Web pages.

Despite often supporting both mechanisms, many weblog engines favor the use of tags instead of categories to classify posts. Wordpress[13] is a blog engine that supports both approaches, and can provide access to its posts and pages through tagclouds.

Some systems were not designed to use tags, but are extended with plugins that add this capability. An example is the Trac software forge[14], and its Tags plugin[15]. Trac integrates several features useful for software development, like a wiki engine, source code browser and issue-tracking system. The Tags plugin allows to label wiki pages, which can then be easily queried to create indexes.

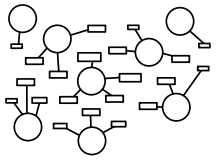
Related Patterns

Folksonomies, like Taxonomies, Thesauri and Ontologies can be used to represent an Index. Like Ontologies, Folksonomies assume an open domain.

If supporting collaboration is more important than a rich description of contents, a Folksonomy makes a better indexing language than an Ontology.

[11] Available at http://www.flickr.com/.
[12] Available at http://www.delicious.com/.
[13] Available from http://wordpress.org/.
[14] Available from http://trac.edgewall.org/.
[15] Available from http://trac-hacks.org/wiki/TagsPlugin.

5.5.8 Controlled Vocabulary



Consider that you want to classify and create an Index for a body of information, to support the readers in quickly reaching the contents they need. The representation of the index, and of the information itself, is done using text. This means it can take on several meanings, but when readers seek the content of a subject, they have a specific meaning in mind.

Example

George is using an electronic address book, where he keeps all of his contacts, including customers, suppliers and friends. The address book allows several kinds of contacts (telephone, email, etc.), but he is using it to store mostly phone numbers.

Today George decided to call his friend Richard Smith, and he was faced with an issue: there are two Richard Smiths in the address book. One of them is surely his friend, and he thinks the other might refer to one of his suppliers, who has the same name. There's also a contact with the name "Ringo" on the list. It's an alternative contact of George's friend, but it's under a name that he didn't remember to check.

Problem

A single term or expression doesn't convey a precise concept, and can be interpreted to different meanings.

When going through a list of terms we read them in the context of the that list, and the way it is being presented to us. A textual expression that is not **semantically rich** may make it difficult for readers to assemble the same meaning that was originally intended. Clearly establishing the meaning of each term will imply a higher **effort** to create and maintain that vocabulary.

Furthermore, two different terms may be intended to convey the same meaning. Such **ambiguity** may make information consumers unsure if two apparently different terms actually mean the same or different things. On the other hand, by representing the same concept with different terms, we are supporting **different information consumers**, which may be looking for the same meaning using different terms.

Solution

Select an official *term to denote each concept, and use it every time you need to refer to that concept.*

Such term is called the *authorized form* of the concept. Pick the most descriptive term (or expression) for the authorized form, making sure that no two concepts share the same term. If the same term is generally used for different concepts, explicitly add a qualifier to the term, to resolve the ambiguity.

Also include aliases of the term in the controlled vocabulary, as *unauthorized forms*, and use them to find their corresponding authorized forms when necessary. When listing the terms of a controlled vocabulary, emphasize the authorized forms, so that the users of the controlled vocabulary can favor them when referring to concepts of the vocabulary. You can show unauthorized forms only on request, and/or format them differently.

Unlike the terms of natural language vocabularies, a CONTROLLED VOCABULARY consists of a list of terms preselected by the author of the vocabulary, which relates the different words that represent the same thing (synonyms) and distinguishes the different concepts with the same name (homographs and polysemes) [ANS05].

A CONTROLLED VOCABULARY reduces language **ambiguity** by establishing a single authorized form for each concept. Controlling a vocabulary requires some **effort**, and some **collaboration** between those involved in its creation, to achieve a shared understanding of the concepts underlying the authorized forms.

Example Resolved

George's address book supports having several contacts assigned to a name, and in order to avoid facing the same problem in the future, George takes some time to reorganize it.

He confirms a few numbers, and starts merging records that refer to the same person. He creates an entry with the name (i.e., authorized form) "Richard Smith", to which he associates his friend's work phone number, and personal phone number. The address book also allows to add aliases (i.e., unauthorized forms), and he adds "Ringo" as an alias for that entry, should he search for that term in the future.

To create an entry for his supplier with the same name, he needs to disambiguate the authorized form using a qualifier, and uses "Richard Smith (suplier)" as the authorized form for that contact.

Known Uses

When using an information system, users often need to fill in fields from a predefined list of possible values, sometimes presented with a combo-box graphical control. The

terms can often be edited only by the administrators of the system, which should choose them carefully to avoid misinterpretation by the users. In this sense, these lists of preselected terms can be seen as controlled vocabularies, even though usually only one form is supported (the authorized form), and it's not possible to express aliases for each term (i.e., add unauthorized forms).

The PSH[16] – Polythematic Structured Subject Heading System – is a controlled vocabulary and an indexing system, built by the Czech Republic's National Technical Library. It was created for indexing and searching contents by subject.

Related Patterns

By eliminating ambiguity and controlling synonyms, Controlled Vocabularies support the creation of Indexes, providing information consumers a better guidance. Some indexing languages, like Thesaurus and Ontology, imply a strong definition of the concepts behind the terms, and are naturally a good fit for the use of a controlled vocabulary. Others use a different approach – Taxonomies and Folksonomies rely on an implicit definition of the underlying concepts through the context in which the terms appear.

5.6 Patterns of Flexible Modeling Tools

As introduced in Sections 3.2.1 and 3.2.3, to create a model is to represent a complex reality in simple terms, focusing on capturing only its relevant aspects. Meta-modeling, in turn, is the use of models to specify other models through the *analysis, construction and development of the frames, rules and constraints to modeling a predefined class of problems* [SV06].

These frames, rules and constraints can be regarded as the *structure* of the contents. By formally (i.e., explicitly) capturing this structure, modelers are expressing information with a higher degree of rigor and objectivity than what would be achieved by capturing free-form contents. Not only is this an advantage from an expressiveness standpoint, it allows to automatically process information in multiple ways, like ensuring its consistency or reusing it in different contexts.

As introduced in Section 3.2.4, despite the benefits of modeling tools, they are still often dismissed in favor of lighter weight approaches that are easier to learn and free users from obeying a model's constraints. Such approaches range from textual

[16] Available at http://www.techlib.cz/cs/564-english-version/.

documents to paper drawings, to white board sketches. The freedom that they allow is especially important during exploratory phases of requirements engineering and design, when creativity and the ability to record incomplete information becomes more important that rigor. The ultimate goal, however, is always to achieve the rigor and objectivity required to build a software system, so it's ironic that the enforcement of the rules and constraints that is inherent to most modeling tools is also the reason why they are so cumbersome to use when creativity and exploratory design is called for.

Flexible Modeling Tools strive to combine the advantages of both approaches, allowing users to trade precision for flexibility, and vice-versa, whenever the occasion calls for it. In the next section you will find an overview of six patterns, and Sections 5.6.2 to 5.6.7 cover the patterns themselves.

5.6.1 Overview

These patterns are addressed to software developers wanting to gain a better understanding of the trade-offs and approaches of creating modeling tools with flexibility in mind. In other words, they can support the creation of modeling tools specifically designed to help their users' work without constraining them in undesirable ways. They may, for example, allow users[17] to pick just the right amount of structure for a given piece of information and allow to change it in the future, or allow to connect elements of a model to related external contents.

Meta-modeling is within the scope of these patterns as, a) meta-modeling is in itself a form of modeling, and b) modeling activities always imply the use of a meta-model, whether intrinsic and implicit (e.g., programmed; enforced by the tool alone) or extrinsic and explicit (e.g., represented using an XML dialect). Domain Specific Languages (DSLs) can also be considered within the scope of these patterns, as their creation and use are in fact meta-modeling and modeling activities using a specific type of representation. We have, however, deliberately not addressed any issue particular to how models may be represented (e.g., graphically or textually), as well as other concerns that are not specific to modeling *flexibility*. Additionally, flexibility is a function of how model and meta-model constraints are approached by developers, and the implications of that can span several different facets of a modeling tool, from user-interaction, to domain validations, to persistence. These patterns don't go into

[17] We often refer to *users*, and it is perhaps worth clarifying that, in the context of these patters, they are the *modelers* – i.e., the end-users of the modeling tools – who are often also software developers.

the details of how they may influence these facets of a modeling tool, but they could certainly be extended into a wider *pattern language* covering such concerns.

The *flexible modeling* principles described by these patterns are applicable to a wider scope than what may strictly be considered a *modeling tool*. Even if not always viewed in these terms, all representations of structured information may be regarded as a model, and the tools that allow creating and maintaining them can be seen as tools supporting *modeling activities*. In this sense, these patterns may be useful to developers creating any sort of tool focused of manipulating representations of structured contents. A map of the six patterns that will be introduced is depicted in Figure 5.4.

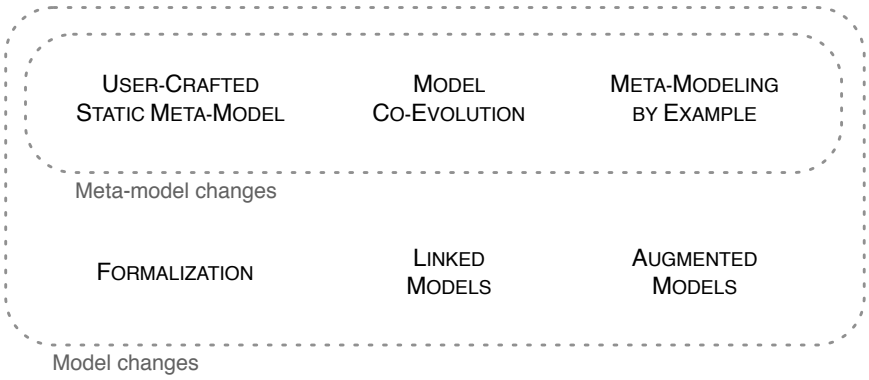


Figure 5.4: Pattern map of the flexible modeling patterns, grouped by the *meta-level* that is the focus of flexibility.

User-Crafted Static Meta-Model (p. 103) – Supports the creation of a meta-model by the modeler himself, before the model is built.

Model Co-Evolution (p. 104) – Explains how models can be evolved together with the meta-models that they depend on, when these meta-models need to change;

Meta-Modeling by Example (p. 106) – Allows to create higher abstraction levels (e.g., meta-models) by giving examples at lower levels (e.g., models).

Formalization (p. 108) – Supports deriving formal/rigorous representations (e.g., models) from information that was firstly captured in informal/non-rigorous ways;

Linked models (p. 110) – Provides a way to connect different but complementary models flexibly;

AUGMENTED MODELS (p. 112) – Supports complementing or annotating models with contextual information that is often not structured or is loosely structured;

Figure 5.4 also shows, for each of the patterns, which meta-levels are subject to change. Other criteria can be used to group the patterns – Figure 5.5 depicts which patterns focus on relaxing or leveraging the constraints between the model and meta-model levels, and which ones focus on doing the same for the relations between models and other, external, contents.

Even though the end goal of these patterns is mainly to support the creation of models, USER-CRAFTED STATIC META-MODEL, MODEL CO-EVOLUTION and META-MODELING BY EXAMPLE directly support creating or introducing changes at the meta-model level. As depicted by Figure 5.5, together with FORMALIZATION, they have in common the goal of providing flexibility between the two modeling levels.

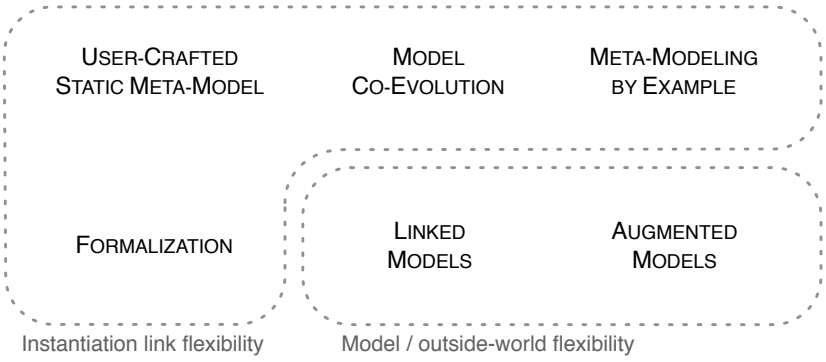


Figure 5.5: Pattern map of the flexible modeling patterns, grouped by the *link* that is the focus of flexibility.

FORMALIZATION, LINKED MODELS and AUGMENTED MODELS differ from the other three patterns in that they focus on the introduction of changes only at the model level. Finally, and not explicitly shown by the figures, LINKED MODELS can be said to be a form of supporting AUGMENTED MODELS, as the contents used to *augment* the model are, in fact, elements from another model.

These patterns were mined mainly from the body of works discussed at the latest three editions of the FlexiTools workshop series[18] [OvdHS+10b, KOvdHS10, OvdHS+11]. Rather than adopting a specific view of what makes a modeling tool *flexible*, they try to reflect the current understanding of the area as taken by its community. Some works in particular have already tried to categorize challenges and approaches taken in this area, and influenced the recognition of these patterns – the very complete

[18] Workshop on Flexible Modeling Tools

summary of the FlexiTools workshop at SPLASH 2010 [19] by Kimelman and Hirschman [KH10], a work by Gabrysiak et al that proposes a classification of meta-models' usage scenarios [GGLS11] and another work by Kimelman and Hirschman [KH11] that includes different interpretations of the notion of modeling tool flexibility.

Some of the authors' previous experience have also influenced the identification of these patterns; namely, their work on Adaptive Object-models [FCW08, FCYA10] and on Adaptive Software Artifacts [Cor10, Cor13].

5.6.2 User-Crafted Static Meta-Model

Modeling tools support their users in expressing information in a given domain or domains. The creation of a modeling tool must consider which domains the modeler will need to address and how expressive they will need to be on those domains.

Problem

The modeling tool developers may be unable to anticipate the domains and expressiveness that the modeler will need.

Conceiving modeling tools and their underlying meta-models requires good abstraction **skills** and considerable **effort**, making it compelling to **reuse** them rather than creating multiple ones, tailored to each specific context. Creating a modeling tool that adheres to a specific meta-model rather than supporting multiple ones also contributes to keeping the modeling tool **simple**.

On the other hand, if the abstractions provided by a modeling tool and its underlying meta-model are not suited for the intended domain, the user might not be sufficiently **expressive** in that domain. In such a case, using that meta-model might be impossible or imply more **effort**, and the resulting model might not always be **reliable**.

Solution

Allow the meta-model to be refined by the modeler, before a model is built.

Use this pattern when the user will need to model domains that can't be fully anticipated by the modeling tool developers. Instead of developing a modeling tool that uses only one specific meta-model and addresses a specific domain, allow the user to supply her own meta-model, tailored to a domain's needs. This means that modeling tool developers must resource to a meta-meta-model to define which meta-models

[19] SPLASH 2010 – Systems Programming Languages and Applications: Software for Humanity

may be supplied by the user. Often, the modeling tool may itself support the creation of the meta-models, in which case it will have a simpler design if the same mechanism is used to support the creation of both modeling levels (i.e., if EVERYTHING IS A THING [FCYA10]).

This pattern can also be referred to as *User-generated Meta-model* [GGLS11].

Known Uses

USER-CRAFTED STATIC META-MODEL is perhaps the *least flexible* of this set of patterns and can be found on some *traditional* modeling tools. For example, the Eclipse Modeling Framework (EMF) supports defining a meta-model and, together with the GMP framework (Graphical Modeling Project), it allows defining a graphical representation and build a complete modeling tool for that meta-model.

MetaEdit+ [TPK07] is an environment that allows creating new modeling languages. It uses graphical meta-modeling to support the early stages of language creation, to define the key language concepts and rules. The meta-modeling language is defined as one of the several domain-specific languages supported by the platform.

Related Patterns

USER-CRAFTED STATIC META-MODEL differs from the other patterns flexible modeling patterns described in this chapter because the flexibility that it allows is only possible until the meta-model is instanced into a concrete model. Namely, it doesn't address meta-model changes after a model has been created. When modelers need the meta-model to evolve after it has been instanced, they need support for MODEL CO-EVOLUTION or META-MODELING BY EXAMPLE.

5.6.3 MODEL CO-EVOLUTION

USER-CRAFTED STATIC META-MODELS are created before modeling activities take place, and they allow modelers to define how expressive they will be able to be during such activities. To use that pattern effectively, modelers must have a very concrete idea of the domain that they will address *before* starting to model in that domain, as the modeling tool might not easily allow refining that idea after the initial creation of a meta-model takes place.

Problem

Modelers may be unable to anticipate the expressiveness that they will need.

Models, and their semantics, directly depend on the constructs defined by the associated meta-models. Users may need to **change** the meta-model during the creation of a model, but they also want models' **consistency** towards their meta-model to be kept. Modelers don't want to spend a lot of **effort** manually maintaining this consistency if they can avoid it. Modeling tools often help by preventing some operations that would cause inconsistencies, but this may be limiting, as it means that the meta-model cannot be be **freely evolved** to all states that it once could, before there were models that used it.

Solution

Allow to change meta-models and automatically evolve dependent models accordingly.

Support Model Co-Evolution when meta-models need to evolve even though models that are based upon them have already been created and the user needs these models to be kept in-sync with their changing meta-models.

Often the operations that are made available to the user at both levels – model and meta-model – are defined by a set of distinct classes that confines the changes that the modeling tool supports, thus applying the Command pattern [GHJV95].

To make the changes at the *meta-model level* have the right repercussions at the *model level*, operations executed at the meta-model level will spawn the execution of other operations at the model level. For each kind of meta-model change, the tool must know how to make the consequential changes at the model level, and sometimes it may require the modeler to provide additional instructions on how the model should be transformed.

The key principles of this pattern can also be applied between the *data* and *model* levels – changes to the model can trigger changes to any data that complies to it.

Known Uses

Gabrysiak et al emphasize the need for modeling flexibility and propose a tool that combines a) a minimization of the restrictions imposed by the meta-model to the model with b) the co-evolution of models as a result of introducing changes to their meta-models [GGS10, GGLS11].

Some approaches and tools to support the evolution of models upon the introduction of changes to their meta-model have been proposed, including Wachsmuth's [Wac07] and Cicchetti's [CDREP08] works.

Some object-oriented frameworks like RubyOnRails[20] (RoR) and Django[21] support the creation of a domain model that is persisted using an ORM. The creation of this domain model usually doesn't involve the use of a modeling tool, nor is its meta-model subject to changes, but the model can, and often does, change during the development of a system. Existing data that complies with such models needs to be changed accordingly, and these frameworks very often provide the mechanisms to do so (e.g., *Migrations* in the case of RoR and *South*[22] in the case of Django).

Related Patterns

Like META-MODELING BY EXAMPLE, and unlike USER-CRAFTED STATIC META-MODEL, MODEL CO-EVOLUTION supports evolving the meta-model after it is instanced.

Co-Evolution is a general concept that may be applied to other domains. Namely, it is often used in the context of software documentation as described by the CO-EVOLUTION pattern [CFFA09b].

As mentioned above, this pattern is also applicable between the model and data levels. In particular, the MIGRATION pattern [FCW08] supports co-evolution between a model and existing data that complies to that model, in the context of ADAPTIVE OBJECT-MODELS [FCA09].

5.6.4 META-MODELING BY EXAMPLE

Both modeling tool developers and the modelers themselves may be unable to anticipate the expressiveness that the modelers will need. To use USER-CRAFTED STATIC META-MODELS effectively, modelers must have very concrete ideas of the domain that they will address *before* starting to model in that domain. MODEL CO-EVOLUTION opens the possibility to refine those ideas after the initial creation of a meta-model takes place, but modelers often discover the new directions that the meta-model will take when trying to express a model and exploring their options at that abstraction level.

[20] Available at http://rubyonrails.org/.
[21] Available at https://djangoproject.com/.
[22] Available at http://south.aeracode.org/.

Problem

The need to change the meta-model diverts the modeler from the creation of the model.

The need to constantly update the meta-model to allow for more expressiveness at the model level diverts the modeler from her main **stream of thought**. Moreover, the need to create or update a meta-model is often a **barrier to entry**, especially when modelers don't have the required **higher abstraction skills** or **technical skills**. This barrier should be as small as possible, but the modeling tool shouldn't be simplified to the point of loosing the **rigor** and **consistency** that a meta-model can support.

These difficulties are easily felt when end-users are asked to collaborate in the design and implementation of a new DSL, for example.

Solution

Build a meta-model by providing examples at the model level.

Support Meta-Modeling by Example when users need to create a meta-model but it's more feasible to start by exploring options and experimenting at the model level than to keep models consistent with their meta-model at all times.

A model always needs an underlying meta-model, but the meta-model doesn't *have to* be explicitly captured before modeling is done – it may exist only in the modeler's mind. By relying on an explicit meta-meta-model, and making minimum assumptions about the meta-model level, modeling tools may support the creation of models in a very unconstrained way, and help modelers decide later what a compatible meta-model could look like.

On this later stage, the modeling tool can automatically infer a meta-model compatible with a given model or models, or it can use them to guide the creation or update of a meta-model when the user wishes to create/update it.

Although not within its main focus, the key principles of this pattern can also be applied between the *data* and *model* levels – a possible model can be inferred from information that has been structured in an *ad hoc* way, in the same way that a possible meta-model can be inferred from a given model.

This pattern can also be referred to as *Lazy Meta-Model* [GGLS11] in the sense that the meta-model needs only to be created when absolutely necessary, or as *Bottom-Up Meta-Modeling* [SCDLG12] in the sense that it encourages modeling to start from lower abstraction layers (i.e., the *bottom*).

Known Uses

Cho et al [CSGW11] tried to make the creation of Domain-Specific Modeling Languages more accessible to domain experts, by using examples of the language provided by the end-users themselves to infer the language's meta-model and semantics. This approach is referred to as *Modeling Language Creation By Demonstration*.

Kuhrmann has developed similar work [Kuh11], striving to better support language engineers in the hard, time-consuming, and knowledge-intensive task of creating meta-models, models and DSLs. The *Process Development Environment* platform allows a free-form language design, and allows to visually represent domain entities and simple associations that are used to derive a meta-model definition.

Smart Office Tools [DOS10] support a content model, capable of visually representing the domain knowledge as a domain diagram, but without tying the user to a specific meta-model. The user is able to customize the diagram notation by creating *styles*, which she can at any time annotate to formalize a meta-model.

Related Patterns

META-MODELING BY EXAMPLE always depends on the creation of a model, from which higher modeling levels are manually built or inferred. Such inference or creation of information constructs out of other pieces of information is something that this pattern has in common with FORMALIZATION. They are, otherwise, very different patterns, as FORMALIZATION acts only at the model level – it can be used when information is initially void of any explicit form of domain structure and only later is it made into an explicit model.

5.6.5 FORMALIZATION

If the modeler needs to change the meta-model and the modeling tool supports MODEL CO-EVOLUTION or META-MODELING BY EXAMPLE, she is able to introduce those changes without being constrained by the existing models. Other times, the modeler won't feel the need to change the meta-model but the instantiation link (i.e., the link between model and meta-model elements) may still be limiting in freely creating the model.

Problem

It is not always efficient, or even possible, to capture information as a model right from the beginning.

Modelers have various reasons for not using formal modeling tools, and rather often resource to free-form tools instead. Despite their **usability**, and advantages for **communication** and **creativity**, these tools don't offer any support for **maintaining** the results as models. Formal modeling tools provide such support, by enforcing conformance to a specific meta-model, and they confer the **semantics** needed to interpret the models objectively.

Solution

Derive formal/rigorous representations from information that was firstly captured informally/non-rigorously.

Support FORMALIZATION when the user's intent is to create a model that complies to a specific tool or to a well-known meta-model but she can't, or doesn't want, to be always constrained by that meta-model. FORMALIZATION frees the user from some or all meta-model constraints during modeling activities, easing the capture of the flow of thought or allowing to experiment and explore multiple options at the model level. The direct result of such a process, whilst not a model, can subsequently be formalized into a model.

FORMALIZATION always starts with free-form information. The move from free-form contents to a model can sometimes be done automatically by the modeling tool. For example, when sketching a diagram, the tool may store a combination of drawing gestures and a resulting raster image to infer what model elements the user meant to represent. The richer the information obtained through the users' input mechanisms, the easier may be to infer the semantics of what the user intended to express. Extraction techniques like image analysis or natural language processing may be used to support the move from free-form contents (e.g., a piece of text or a raster image) to a model. Additionally, when it's not possible to automatically infer a model, the modeling tool can interactively assist the user in manually building it from the contents.

Known Uses

The SKETCH API [SB10] allows developers to add sketch recognition to modeling editors built for the Eclipse IDE. It leverages touch-enabled devices and their great potential as drawing tools, allowing to create and manipulate freehand sketches from which a formal model can be inferred and associated with an underlying meta-model.

Architects Workbench (AWB) [ABK+06] provides totally free-form text entry and the ability to evolve them towards formally structured contents, maintaining the

traceability from one form to the other. AWB's users can use a *markup and model* technique to create model elements directly from the text.

UNICASE [HNA+10] supports explicit traceability between information with different levels of abstraction and between loosely-formal *project models* (the artifacts that describe a software project, such as tasks, bug reports and informal communication) and *system models* (the artifacts that describe the system, such as functional requirements, UML models and detailed system specifications). This traceability information is then used to support the formalization process that underlies the propagation of changes from project models to system models.

Related Patterns

Formalization is similar to Meta-Modeling By Example to the extent that both patterns relax the constraints between the model and meta-model levels and help the modeler create new information constructs, or even automatically infer such information constructs, from other contents. Otherwise, they are very different patterns, as Formalization supports creating a model, and Meta-Modeling By Example supports introducing changes at both modeling levels – model and meta-model.

Organizing documentation towards Domain-Structured Information [CFFA09b] can be seen as a light attempt to Formalization, as the contents are organized according to their domain but not to the extend of becoming a model.

5.6.6 Linked Models

User-crafted Static Meta-model, Model Co-evolution and Meta-modeling By Example allow modelers to increase their potential expressiveness at the model level by enriching the meta-model. Sometimes this means increasing the scope of the meta-model, which may make it difficult to manage or even overlap other readily available meta-models that would be perfectly suited for modeling that part of the domain.

Problem

Models are conceived for specific domains, but modelers may need to address a broader domain than each model is able to address individually.

Wanting to be **expressive** in a certain domain, software developers sometimes resource to creating different models – for meta-models **specifically tailored** to different

parts of that domain – or to providing different **views** over the same parts of the domain. However, these meta-models are not necessarily designed for **interoperability**, which implies a semantic gap between the produced models.

Solution

Allow two models or domain-specific languages to be linked as needed.

Support Linked Models to enable users to connect different but complementary models in a flexible way. This pattern allows to compose models specialized to different parts of the domain.

Let the user of the modeling tool define, at the meta-model level, how the models can be linked or, instead, let her connect model elements in an *ad hoc* fashion (i.e., leave it entirely to the her choice which model elements can be connected, as appropriate). The best approach – meta-model-based or *ad hoc* – will depend on how the two domains relate to each other and on the amount of flexibility that the modeling tool is intended to provide. On both cases, this pattern may also be referred to as *Multiple Meta-Models*, in the sense that you may regard the result as a single set of connected model elements (i.e., a single *model*) that comply to more than one meta-model.

Known Uses

OMME (*Open Meta Modeling Environment*) [VJ10, VZJ11] is a meta-modeling environment implemented on top of the Eclipse platform that provides both textual and graphical notations. Among other features, this environment allows to represent and link models of arbitrary kind – such as, a process model and a data model – allowing to choose the models which fit best in a given situation.

Chiprianov et al propose a language tool for telecommunication network designers that provides a partial syntactic and semantic automatic interoperability between different languages, corresponding to different viewpoints used in the definition of a telecommunication service [CKR10].

Microsoft Visio and similar tools allow users to choose between multiple *stencils*, thus allowing to combine modeling elements from multiple sets (i.e., multiple meta-models). In practice, the connections established using this approach arbitrarily link different models, while their meta-models remain independent. Despite its flexibility, the communication within a team using such models is sometimes more difficult than using completely meta-model driven models, as the semantics of the connections

between these different elements is not defined by a meta-model, as highlighted by Gabrysiak et all [GGLS11].

Related Patterns

Linked Models can be seen as a form of Augmented Models where the contents used to augment the model are, in fact, other models or elements of other models.

When the meta-model doesn't establish how the models or model elements are to be linked, Meta-Modeling by Example can be used to allow modelers to explicitly create such rules at the meta-model level after links have been created.

Even though Information Proximity [CFFA09b] addresses software documentation in general, Linked Models relies on the same principles, as does Augmented Models.

5.6.7 Augmented Models

Models hardly exist in isolation. Modelers may be able to Link Models to go beyond the scope of a single meta-model, by connecting a model to other modeled contents. Sometimes, though, some of the contents that the modelers would like to relate a model to are not themselves a model.

Problem

Models are only a part of the information that needs to be captured and they need context to be fully understood.

Extending and tailoring the meta-model to the modeler's specific needs may be the most immediate solution when more expressiveness is needed at the model level, but supporting meta-model changes implies added complexity in the development of the modeling tool and the underlying meta-model. Modeling tools should be as **simple** as possible to develop, but also to be able to **express** as much detail as possible. Moreover, the information may be vague and difficult to represent as a model, if at all possible. Although we want information to be **structured** as much as possible – after all, the goal is to support the creation of models – we also want to support the capture of all **valuable** information independently of their level of structure. Free-form contents can contextualize, and thus allow to better **understand**, the created models.

Solution

Complement or annotate models with contents that provide additional context.

Support AUGMENTED MODELS to allow users to complement or annotate models, often with contents that are not structured or are loosely structured. The extra contents can be added at several granularity levels – they can be added to the model as a whole, or to specific model elements. In the later case, similarly to LINKED MODELS, the model elements to which new contents are added can be constrained at the meta-model level, or the modeling tool can allow them to be added arbitrarily to any model element.

Due to the lack of formal information constructs, it may be impossible to process the extra contents in any meaningful way, but they can be very useful as contextual information, required for the model to be better understood and be made use of.

Known Uses

Literate Modeling was born by taking the idea of Literate Programming beyond source code. Instead of weaving only textual descriptions with source code, they are also combined with UML models. In a way, models are *augmented* with textual descriptions that contextualize them and support their understanding by domain experts that are not proficient with modeling languages [Job93, AEQ99].

The work by Breitman et al [BPB10] acknowledges that models are currently unable to accommodate all the levels of knowledge (from vague to concrete) that modelers need to represent, and proposes the creation of model *narratives*, as annotations in the form of graphics, audio, video, URLs, and other informal representations that are possibly vague and incomplete.

Nagel et al have proposed a solution to bridge the gap between informal communication and formal project model elements [NHKN10]. They address the capture of audio conversions within their specific context, allowing to find specific information in a large number of recordings.

Related Patterns

AUGMENTED MODELS can be seen as a more general case of LINKED MODELS, where the contents being linked are not necessarily other models or elements of other models.

Model annotations can be free of any structural constraints, but it may be possible to employ FORMALIZATION to make their underlying structure explicit and make those contents part of the model itself. If this may result in new kinds of model elements,

then META-MODELING BY EXAMPLE may too be used to define them at the meta-model level.

Even though INFORMATION PROXIMITY [CFFA09b] addresses software documentation in general, AUGMENTED MODELS relies on the same principles, as does LINKED MODELS.

5.7 Patterns of Adaptive Object-Models

The ADAPTIVE OBJECT-MODEL (AOM) architectural pattern was introduced in Section 3.2.5, which has presented the core patterns to understand how AOMs are designed and built. The endeavor taken by Yoder et al. [YFRT98] and later by Welicki et al. [WYWJ07] to map a pattern language for AOMs outlined a useful overview of this architecture and helped to document it more systematically. We have continued this work and contributed additional patterns to the larger body of knowledge on AOMs. The language with the new patterns was reclassified taking the categories defined in previous works as a starting point [FCW08, FCYA10, Fer10] and is presented in Figure 5.6.

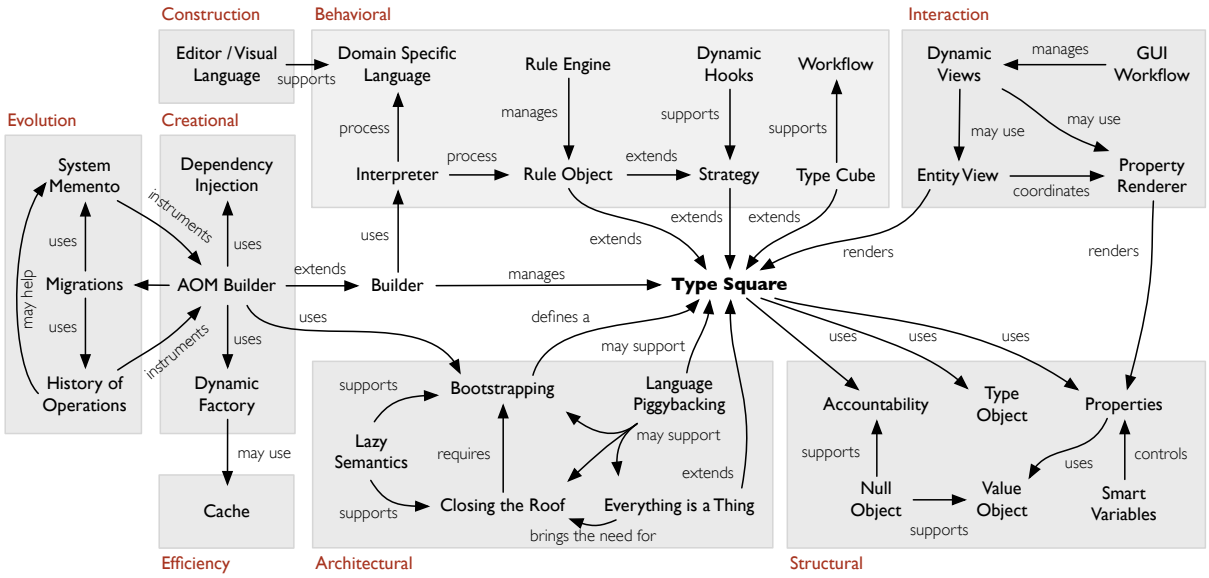


Figure 5.6: Pattern map of the adaptive object-models patterns.

The patterns on the *architectural* and *evolution* categories are contributions of this work and, respectively, support defining an infrastructure and creating the mechanisms for managing the introduction of changes to the model and meta-model. These patterns won't be extensively detailed in this thesis, but the following overview may be important to understanding the role that they play in the design of the reference architecture and implementation (Chapter 7).

Architectural Patterns

Everything is a Thing [FCYA10] – Addresses the problem of having multiple representations of the same underlying concept.

Closing the Roof [FCYA10] – Encloses the structure and meaning of a meta-architecture by stopping the seemingly infinite escalation of meta-levels.

Bootstrapping [FCYA10] – Addressed the fact that any enclosed structure able to define itself relies on a (small) set of basic definitions, upon which it can build more complex structures.

Lazy Semantics [FCAY11] – Defers the meaning of a definition until it is absolutely needed, to avoid that structural elements may depend on their own definitions.

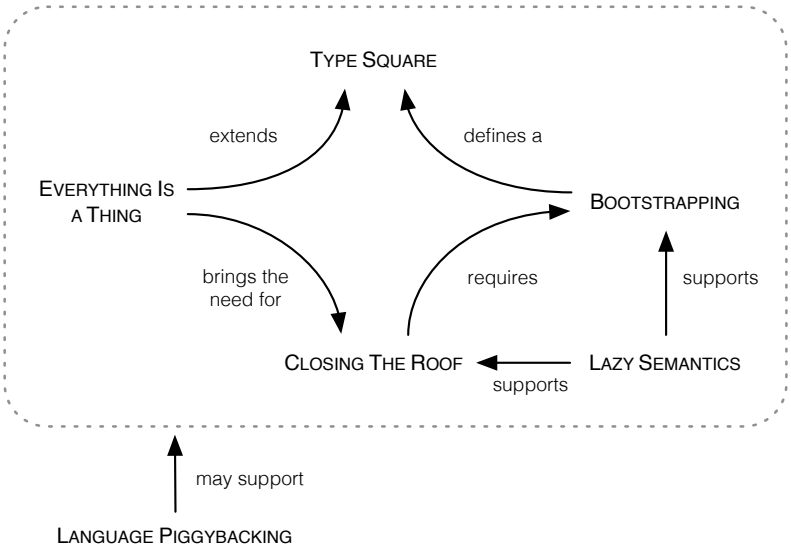


Figure 5.7: Pattern map of the adaptive object-model architectural patterns.

LANGUAGE PIGGYBACKING[23] (Appendix A) – Uses a programming language's reflection mechanisms to avoid reimplementing constructs that already exist in the language.

Evolution Patterns

HISTORY OF OPERATIONS [FCW08] – Addresses the problem of maintaining a history of operations that were made upon a set of objects.

SYSTEM MEMENTO [FCW08] – Deals with preserving the several states the system has achieved throughout its evolution.

MIGRATION [FCW08] – Addresses the concern of performing evolution upon the system while maintaining its structural integrity. It relies on the patterns HISTORY OF OPERATIONS and SYSTEM MEMENTO to ensure enough information is gathered.

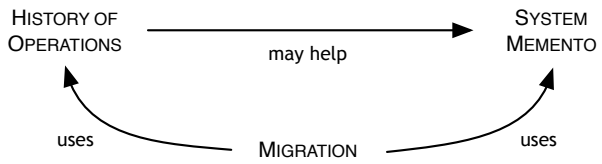


Figure 5.8: Pattern map of the adaptive object-model evolution patterns.

5.8 Summary

In brief, the patterns introduced in this chapter document solutions that play a part in this research. Namely, they formalize good practices and designs on software documentation, information classification, flexible modeling tools and adaptive object-models. Documenting this knowledge contributed to the design of the Adaptive Software Artifacts approach (Chapter 6) and to the development of the Adaptive Software Artifacts Plugin for Trac (Chapter 7).

[23] Unlike the other AOM patterns described in this chapter, LANGUAGE PIGGYBACKING hasn't been published before and therefore is included in Appendix A. This pattern hasn't gone through a shepherding process and review in the context of a writer's workshop, so its maturity should not be considered on par with the remaining patterns presented or mentioned in this work, and we should be cautions when presenting it as a *pattern*. This is clear by the *known uses*, which include solely the reference architecture and implementation that is introduced in Chapter 7.

These patterns use different abstraction levels but don't try to cover the entire problem and solution spaces of the approach. Many patterns could still be added. Two examples of such additions could be Bottom-up Information Structure and Top-down Information Structure, which approach some of the issues already described by the flexible modeling tools patterns (Section 5.6) at a more abstract level. Many more could also be added under the topic of adaptive object-models, starting with those already mapped in Figure 5.6 that were not yet described.

Chapter 6

The Adaptive Software Artifacts Approach

Throughout a project's life, different software artifacts are created and evolved. They take part of the sense-making process in which team members identify recurring information structures that underlie a given body of knowledge. The team may need to capture, share and reason about the ideas in that body of knowledge to discover how they can be structured, therefore they may first capture them as free-form contents, like text documents and, only afterwards, capture them as increasingly more specialized artifacts, such as task descriptions, models and source code.

On the one hand, capturing structure explicitly makes information more concrete, unambiguous and terse. On the other hand, free-form contents have the benefit of not being subject to structural constraints, which is of importance during exploratory work. Other differences are that free-form contents don't directly support sharing information structure between team members and information is not easy to automate – e.g., the cost of maintaining free-form contents is high, as keeping their consistency requires continuous review. Moreover, organizing and classifying information for efficient access is often difficult and classification schemes may also need to be constantly updated to reflect the evolving body of information.

The Adaptive Software Artifacts approach is described in detail in the following sections. It combines the benefits of free-form and structured contents with the objective of making information within software development teams easier to use and evolve, especially in the context of medium-to-large projects, where the amount of knowledge involved easily heightens these concerns.

This chapter starts by describing the approach itself (Section 6.1) and goes on to describe respectively the design principles and the concrete activities of the approach

(Sections 6.2 and 6.3). It concludes comparing the Adaptive Software Artifacts approach is with other similar approaches (Section 6.4).

6.1 Approach Concerns and Goals

Traditional software artifacts constrain information by enforcing a set of rules, which bind artifacts to a pre-determined information structure defined by the kind of artifact in question. The Adaptive Software Artifacts approach is designed with flexibility in mind, and enables to: *a) create user-defined types of artifacts* – i.e., information with a custom-tailored structure, to fit the specific project's needs; and *b) change such types of artifacts*, to better support the knowledge evolution needs of the project – these artifacts are adaptive in the sense that their attributes and relations with other artifacts don't need to be established from the start and can be freely and easily evolved by the users. Information based on adaptive software artifacts is not as bound to strict constraints like other structured artifacts usually are.

Furthermore, the benefits over the traditional dichotomy between *structured* and *free-form* contents extend beyond the support to expressing *ad hoc* knowledge structures explicitly. The approach allows the consistency of the contents to be more easily maintained, by making it easier to see which topics are common across free-text documents and by comparing contents with their expected structure; and it supports access to the text contents through a classification scheme that is dynamically built from the connections between the text contents and the adaptive software artifacts.

These goals can be described in the terms of the concerns introduced in Section 4.3.

C1. Expression of information structure

Structure may come to information through a top-down process. Consider when someone capturing knowledge knows, at the outset, how that particular piece of information can be structured. Her mental model of that knowledge may be close enough to a specific kind of software artifact that she is familiar with, and the decision to create an artifact of that type will not need much thought.

But structure may also come to information through a bottom-up process. There may, initially, be a lot of uncertainty as to how the contents can be structured – the mental model for that piece of knowledge may still be very fuzzy and none of the more structured kinds of software artifact be a good fit. Or it may be the case that knowledge is still evolving quickly, and the software artifacts that would fit it are not flexible

enough to keep up easily with that evolution. In such cases, information capture tends to be avoided altogether until a later stage, or start being done as free-text or other weakly-structured form. It gets captured as more specialized kinds of software artifacts only when the mental model for that particular piece of knowledge becomes clearer or its rate of change is low.

Adaptive Software Artifacts support explicitly capturing (and sharing) information structure, allowing contents to be understood (or *consumed*) more easily. It does this by trying to combine the best of the two processes: team members can use a top-down process by defining new types of software artifacts, with their own specific attributes, and to instance them as needed; and they are also able to use a bottom-up process, and incrementally add structure to the textual contents, as new knowledge is being acquired, thus making new software artifacts to gradually emerge from the text contents. Commonalities can be identified in these emergent artifacts, at which point new types of artifacts can be explicitly defined.

Most tools that allow capturing any kind of software knowledge support a top-down process. With the Adaptive Software Artifacts approach users are able to mix and match the two processes in an integrated environment such as a software forge. Additionally, independently of the process through which the artifacts come to be, they can be evolved without being subject to the hard rules that *non-adaptive* software artifacts are bound to, as it may be expected that their structure complies to a specific type of artifact, but such compliance is not necessarily required.

C2. Consistency maintenance

By easing the creation of structured contents, the Adaptive Software Artifacts approach opens the possibility for several use cases. By creating adaptive software artifacts from the free-text contents, we are both identifying and abstracting the key topics of the text and connecting the resulting abstractions to their respective documents. This allows us to assess if introducing changes to a free-text document might have an impact on the consistency of others – when changing a document about a certain topic it may be brought to the user's attention what other pages refer to the same topic. Additionally, when contents are structured and associated with a type, one may assess their consistency towards their type.

Easing consistency maintenance means easing the *evolution* of existing contents and the *creation* on new ones.

C3. Classification of the contents

Looking for contents within a large collection of free-text documents can be difficult, especially when the information needs are ill-defined. But the contents of an Adaptive Software Artifact can be created organically from the textual contents, and they may be seen as descriptions of those textual contents. Therefore, they can be used effectively as an indexing mechanism and a rich classification scheme can be automatically derived from them. Such an always-updated index has a low cost of *creation* and *evolution* and allows information to be more easily found and thus *consumed*.

6.2 Design Principles

This approach assumes the development of tools and environments that support it. Chapter 7 presents a reference architecture and implementation of such a tool, the development of which was based on the set of design principles or high-level requirements introduced in the following paragraphs.

Wiki Design Principles. The success of wikis as collaborative authoring platforms owes much to the set of principles used to design them[1]. These principles show themselves important in the context of wikis but are abstract enough to be used to design other contents-creation tools. They don't directly address any of the main goals and research problems described in the previous sections of this chapter but take into account the collaborative nature of software development.

Integrated Environment. Derives from acknowledging the need to make new development tools available within an environment already familiar and used by software developers and to take a more holistic approach to knowledge capture. An Integrated Environment (p. 76) supports the maintenance of consistency (C2, p. 51 and 120) by keeping related contents of different natures easily accessible from each other.

Domain-Structured-Information. This principle is illustrated by the Domain-Structured Information (p. 74) pattern, but also by the Controlled Vocabulary (p. 97) pattern. It helps to express and organize contents according to their domain (C1, p. 51 and 120). Indirectly, it contributes to Information Proximity (p. 66), which helps Co-Evolving (p. 71) related documentation fragments, and thus preserving their consistency (C2, p. 51 and 121).

[1] The design principles of wikis are described in more detail in Section 2.3.2.

Structure Co-Evolution. This principle is strongly inspired by MODEL CO-EVOLUTION (p. 104) and to some extent by the more general CO-EVOLUTION (p. 71). It consists of the mechanisms to evolve structured contents while maintaining them consistent (C2, p. 51 and 121).

Flexible Structure. The *bottom-up* expression of structured information (C1, p. 51 and 120) implies flexibility requirements that can be enunciated as FORMALIZATION (p. 108) – adding structure to free-form contents – and as META-MODELING BY EXAMPLE (p. 106) – identifying the structure commonalities of a body of contents, and make those commonalities explicit.

Automatic Index. Improving the access and classification (C3, p. 51 and 122) can be addressed through a combination of an INDEX (p. 82), a CONTROLLED VOCABULARY (p. 97) and DOMAIN-STRUCTURED INFORMATION (p. 74).

6.3 Activities

The approach may be decomposed as a specific set of activities. These activities are described next in this section and are summarized and illustrated at an abstract level in Figure 6.1. They should be used as a set of concrete feature requirements when developing tools to support the approach.

6.3.1 Creation

Creation activities are those focused on the capture of structured contents. Namely, the environment should support the creation of new kinds of software artifact by developers. Such artifact types are not pre-determined during the conception of the environment, so they won't provide some of the most specialized behavior of any built-in artifact types that the environment may offer, but they will allow the users to specify their own data-centric structures. Meta-modeling techniques may be used to support the definition of such artifacts, their properties and the relations to other artifacts.

A1. Create artifacts and artifact types. This activity allows defining new *artifact types* and new *artifacts*, and specifying their attributes and attribute values. Additional expressive power is given by an *inheritance* mechanism, which allows an artifact type to be reused by another artifact type, and by the possibility of defining *domains* and *cardinalities* for the attributes of an artifact type.

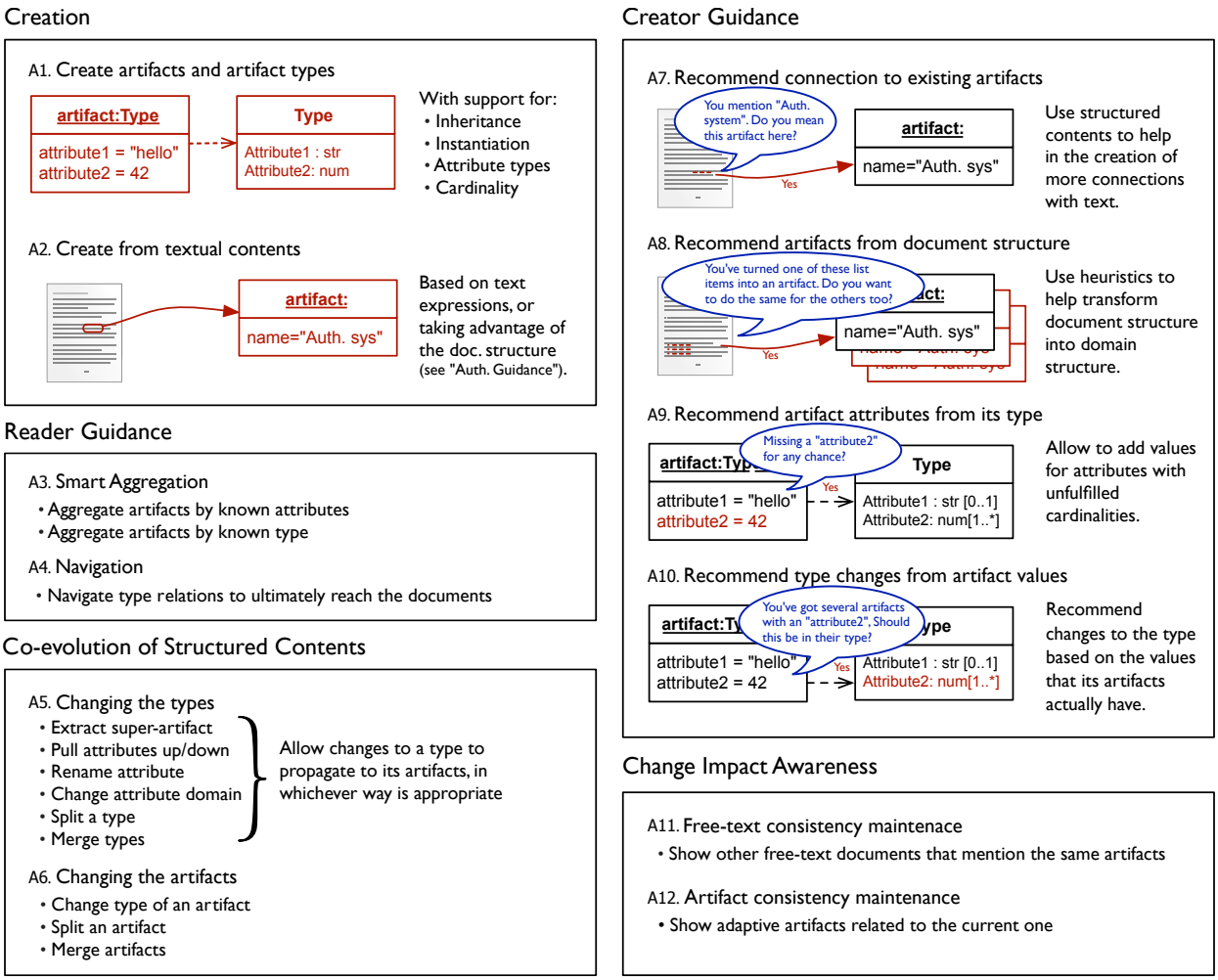


Figure 6.1: Overview of the activities of the Adaptive Software Artifacts approach.

The artifacts created by users may be of an unspecified type, or they may be of one of the artifact types that may have been previously created. In the later case, the connection between the artifact and its type (i.e., the *instantiation link*) is a point of flexibility. In other words, even though an artifact has a type, it doesn't necessarily have to follow the structure that the type defines – it may not have all the attributes defined by the type, or not strictly follow their domain and cardinalities, and may even define its own specific attributes.

A2. Create from textual contents. The creation of artifacts can be done taking textual contents as a starting point. Without ever losing the context of the free-text document, users should be able to create a new artifact from a text selection. That text fragment can be used as the value for a default attribute of the newly created artifact, which may afterwards be completed with more/other attributes and values. The goal is to reduce the effort in the incremental process of providing

more and more domain-oriented structure.

6.3.2 Reader Guidance

These activities address the classification and findability of the contents. Namely, they consist of providing a subject index that is automatically assembled using the contents of the available Adaptive Software Artifacts, and that documentation consumers may use to find the free-form contents that they may need.

> **A3. Smart Aggregation.** Artifacts may be aggregated by their common features, the most obvious of which is their type. Although users may create untyped artifacts, even these will have their own internal structure, such as attributes and relations with other artifacts, so even these can be aggregated by their common structural elements. When a great quantity of untyped adaptive software artifacts exists, each with their own internal structure, this activity helps to identify commonalities in the information structure that may come to be expressed as new artifact types.

> **A4. Navigation.** Users should be able to navigate the subject index built by aggregating the adaptive software artifacts' contents, to find a particular adaptive software artifact that they may be looking for, or to find the set of free-text documents in which it is used. Such a list of aggregated artifacts effectively constitutes a subject-index of the free-form contents in the platform.

6.3.3 Co-evolution of Structured Contents

While having structured contents naturally makes inconsistencies easier to detect, these inconsistencies are not necessarily easier to correct. This is due to some extent to the flexibility supported by the *creation* activities, which frees information capture from some barriers to being evolved but also make inconsistencies easier to surface.

Co-evolution activities have the goal of making structured contents as easy to evolve as free-form contents. They ensure that artifacts and their underlying model will evolve together and are kept consistent. They allow to automatically update related pieces of information so that they are kept consistent, similar to how a refactoring tool supports applying consistent transformations to source code.

> **A5. Changing the types.** Provide information creators a predefined set of transformations useful when changing artifact types, and make these changes

propagate to its artifacts, in whichever way is appropriate. These transformations may draw inspiration from code refactorings. A few useful examples could be:

Extract super-artifact – Split the attributes of an artifact type into two different artifact types connected by an inheritance relation. All the artifacts of the original artifact type will become instances of the new artifact sub-type.

Pull attributes up/down – Move an attribute up/down in the inheritance chain to a artifact super/sub-type.

Rename attribute – Change the name of an attribute of an artifact type and choose whether to cascade that change to the same attributes of all its artifacts.

Change attribute domain – Change the type of an attribute of an artifact type, and choose whether to cascade that change to the same attributes of all its artifacts.

Split a type – Split the attributes of an artifact type into two different artifact types, and decide whether the artifacts of the original type should also be split among the two new types by a chosen criteria.

Merge types – Merge two artifact types into a single one. The artifacts of both types all become instances of the new type.

A6. Changing the artifacts. Provide a set of structure transformation operations at the artifact level. Examples include:

Change type of an artifact – Switch (or remove) the type of an artifact.

Split an artifact – Split the attributes (and respective values) of an artifact into two different artifacts.

Merge artifacts – Join the attributes of two artifacts into a single artifact of the chosen type.

6.3.4 Creator Guidance

This set of activities guides team members in the creation of structure, regardless of whether it is created through a *top-down* or *bottom-up* process. Such guidance is offered as a set of suggestions or recommendations that are given when contents are being created. These activities are inspired by the *Time-shifted Co-evolution* variant of the Co-evolution pattern.

Time-shifted co-evolution does not force consistency. It allows artifacts to diverge from a model that was established for them, but makes developers aware of that divergence, so that they may restore consistency if, and when, they wish to do so. These activities make it easier to maintain the real value of artifacts throughout their lifetime.

A7. Recommend connection to existing artifacts. Identify terms in the textual contents that match existing adaptive software artifacts and suggest the creation of new connections between the two.

A8. Recommend artifacts from document structure. Free-text cannot be said to allow the expression of domain structure but it supports a document-oriented structured made of some elements – such as sections, paragraphs, lists, etc. – that sometimes correlate strongly with an underlying domain structure. This activity must rely on a few heuristics to help transform a free-text document structure into domain structure. For example, if one of the items of a list in a document is made into an adaptive software artifact, the remaining items of the list could likely originate adaptive software artifacts of the same type.

A9. Recommend artifact attributes from its type. The environment does not enforce that artifacts must obey the structure defined by its type but suggests information creators to add values for attributes with unfulfilled cardinalities.

A10. Recommend type changes from artifact values. The environment also suggests changes to the type based on the values that its artifacts actually have.

6.3.5 Change Impact Awareness

These activities focus on assisting information creators in contexts where it is difficult to detect if inconsistencies are being introduced.

A11. Free-text consistency maintenance. When editing a given free-text document, the environment should point out other documents that mention the same artifacts, as it is likely that they address the same topics.

A12. Artifact consistency maintenance. When editing a given adaptive software artifact, the environment should point out other adaptive software artifacts that are related to it, either because they directly refer the one being edited or because they are mentioned by the same free-text documents.

6.4 Comparison to Other Approaches

Wikis bring many benefits to software development but only some variants like *extended wikis* and *semantic wikis* allow expressing information according to domain-oriented structures and, to varying degrees, combining them with the wiki's textual contents. From these, only *semantic wikis* try to support the creation of arbitrary kinds of structure. The Adaptive Software Artifacts approach uses elements from different *semantic wikis*, as it considers a) using text as a starting point for structure creation, b) taking a bottom-up approach to structuring contents – structuring text contents first, and abstracting it into types later and c) capturing recurrent structures expressively (i.e., as distinct artifact types) – using notions like inheritance, attributes, attribute domains, etc. None of the semantics wikis that we know and, in particular, none of the ones described in Section 2.3.2, tries to combine all of these elements.

The description of our approach does not specify how structure and types should be stored. Nevertheless, it is worth noting that the reference implementation detailed in Chapter 7 distances itself from most semantic wikis, in that it does not use the notion of a *wiki page* to define types or other data-oriented contents. This option is in line with some design principles of wikis, namely that a few text conventions should provide all the necessary formatting (*mundane*) and that the formatted output of a page should suggest the input required to reproduce it (*overt*).

Literate Programming and its derivatives (e.g., code annotations, elucidative programming) allow to combine source code with textual descriptions, but they don't delve into structuring these textual contents. Comparatively, the Adaptive Software Artifacts approach encourages and leverages the creation of semantically richer documentation.

The approach uses several **flexible modeling** principles to combine the benefits of free-form and structured contents, even if the end-result of using it is not necessarily a model. It focuses on structuring information of free-text documents (e.g., of wiki pages) into objective and meaningful *elements*, on an as-needed basis. The result is the creation of instances (adaptive software artifacts) and model elements (adaptive software artifacts' types), which sets this solution apart from other *flexible modeling* approaches, which focus on the model and meta-model levels. Moreover, the main goal is not to represent this information as diagrams, or to directly play a part in the creation of executable artifacts, but to support structuring and organizing textual contents.

Our approach can also be compared to current **software forges**. Software forges support creating and relating different kinds of software artifacts, possibly more

effectively than a generic approach such as ours might be able to. Furthermore, these platforms often provide the mechanisms of a framework, allowing new kinds of software artifacts to be added through plugins or similar extension approaches. An important difference, though, is that the Adaptive Software Artifacts approach achieves a different balance between the flexibility of easily creating new kinds of artifacts, and the amount and quality of the features that can be provided to handle specifically those kinds of artifacts. The development of an extension for a software forge is not necessarily easy and expedient, and usually implies a non-negligible cost. Our approach doesn't aim to allow as much, and as good, tool features to handle specific kinds of content but aims to lower considerably the barrier to structure contents that otherwise will remain free-form.

6.5 Summary

This chapter has presented the Adaptive Software Artifacts approach to documenting software, which is designed to combine benefits of free-form contents and benefits of structured contents. Namely, it focuses on providing greater flexibility to developers in what regards adding structure to free-form contents and on leveraging that structure for consistency maintenance and for the classification of the textual information. When compared with other environments used in software development, this flexibility comes at a cost of features that tools could provide to handle new kinds of information (i.e., artifact types) that users may define, but considerably lowers the barrier to structure contents. Contents which otherwise would most probably remain free-form.

The design principles and activities introduced in this chapter were used as the base requirements for the reference architecture and implementation described in Chapter 7 which, in turn, provided feedback that was used to improve the approach.

Chapter 7

Reference Architecture and Implementation

The approach and activities described in Chapter 6 are abstract and developers using them as a basis for creating tools may surely come across more specific issues. This chapter describes a reference architecture and implementation that exemplifies how some technical challenges may be overcome to bring the approach to a development environment. The same implementation was used to validate part of the approach, as is detailed in Chapter 8.

This chapter starts by overviewing the context in which the implementation was created and the extent to which it covers the activities detailed in the previous chapter (Sections 7.1 and 7.2). The Trac software forge was chosen as a base platform for the development and therefore will be briefly described (Section 7.3) before going into the technical aspects of the implementation (Section 7.4). Before concluding, the chapter will make a final analysis of the implementation and its development (Section 7.5) and describe how others may obtain it (Section 7.6).

7.1 Overview

Many of the integrated environments currently available could be extended to support the Adaptive Software Artifacts approach. We believe in the merits of this choice when compared to developing an entirely new tool from scratch. An integrated environment will easily support a wider range of common software artifacts *out of the box*, and allow a better balance between the support for specialized (inflexible) software artifacts and the support for the user-defined information structures enabled by the Adaptive

Software Artifacts approach.

The reference architecture and implementation described in this chapter was built as a plugin to the Trac software forge [Edga]. Most software forges already allow the creation of free-text documents (i.e., wiki pages), the creation of some specific software artifacts (such as tasks, milestones, releases, etc.) and the access to other software artifacts that are external to the software forge *per se* but that are handled within the same environment (like the revisions and source code on a revision-control system). Trac proved to be an interesting platform for the kinds of artifacts that it already supported but, above all, for its extensibility. Our previous experience in developing for this platform made us confident that it would be practicable to use it as a foundation for the approach.

As we have briefly mentioned before, the Adaptive Software Artifacts approach is not specific to software forges, or to Web-based environments, so it is worth noting that the design presented by this chapter goes beyond the generic approach that was described in the previous chapter. This design was necessarily influenced by the Trac environment and the specific extensibility mechanisms that it provides.

7.2 Supported Activities

The development focused first on the set of activities of greater importance to the approach in general and, in particular, to the statistical experiment described in Chapter 8. Table 7.1 shows the extent to which the reference architecture and implementation described in this chapter currently supports the approach. The presented features match directly the activities described in the previous chapter and refer to version 0.5 of the implementation.

The plugin provides two additional features that don't directly contribute to the approach, and that are shown separately in Table 7.2. *F13* is an experimental feature, developed when exploring new evolution directions for the plugin; a few more details about it's usage are provided in Section 7.4.3. For *F14* the motivation was purely the statistical experiment described in Chapter 8 and the data-collection and measurements that it required.

Feature	Support
F1. Create artifacts and artifact types	✓*
F2. Create from textual contents	✓
F3. Smart Aggregation	✓**
F4. Navigation	✓
F5. Changing (refactoring) the types	✗
F6. Changing (refactoring) the artifacts	✗
F7. Recommend connection to existing artifacts	✓
F8. Recommend artifacts from document structure	✗
F9. Recommend artifact attributes from its type	✓
F10. Recommend type changes from artifact values	✗
F11. Document consistency maintenance	✓
F12. Artifact consistency maintenance	✗***

* Attribute types/domains support the *text* and *number* native types. They may in the future be extended to support the software artifacts provided by the platform *out of the box* as well as user-defined adaptive software artifact types.
** Non-typed artifacts are currently aggregated into a single set. This feature can still be improved by taking into account the attributes that such artifacts may have in common to aggregate them into different sets.
*** Direct connections between adaptive software artifacts are already used to show a list of related artifacts when viewing an artifact. A minimal implementation of this feature could be achieved by showing this list when the artifact is being edited. Furthermore, the feature can be improved for consistency maintenance if it additionally includes artifacts referred by the same wiki pages.

Table 7.1: Features of the Adaptive Software Artifacts Plugin 0.5.

Feature	Support
F13. Show Adaptive Artifact's graph as an object diagram	✓
F14. Complete tracking of how the platform is used	✓

Table 7.2: Additional features of the Adaptive Software Artifacts Plugin 0.5.

7.3 The Trac Platform

The Trac software forge features a *wiki*, which software developers can use to create the project's documentation, a *timeline*, where different events tracked by the platform are reported in chronological order, a *roadmap*, where past and future milestones[1] of the project are made available, a *source code browser*, which allows to inspect the source code and revisions maintained by the revision-control system, and an *issue-tracking system*, which allows to manage tasks related with the project. Additionally, Trac provides a *search module* that covers all the contents in the system. The options to access

[1] A *milestone* as understood by Trac is composed by a set of tickets (i.e., tasks), a short description, and a date in which its completion is due.

these features can be seen in Figure 7.1, where is shown an example of a software development project using Trac.

Figure 7.1: A real-world instance of Trac, used for the Wordpress project, showing the home page of the wiki, and the menu options that provide access to Trac's main features.

The platform's architecture is based on a set of pluggable components [Edgb], which make the development of new plugins convenient. The default components supplied with Trac 1.0 provide over fourty extension points that may be used by developers to create new extensions [Edgc]. Such new extensions may, in turn, use the same mechanism to provide their own extension points to third parties, as depicted by Figure 7.2.

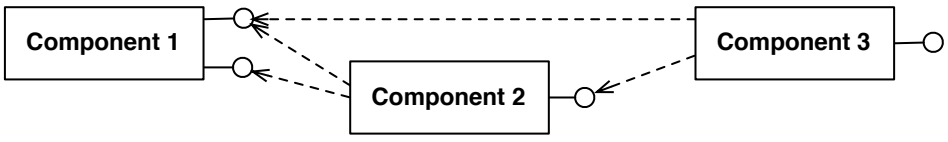


Figure 7.2: Example of Trac's components and extension points, as a UML component diagram.

Trac is made available as open-source under a modified BSD license. The core of the platform and its numerous plugins[2] are built using the python programming language[3].

7.4 The Adaptive Software Artifact Plugin

The extension to the Trac platform was created as a plugin, described by this section.

7.4.1 Architecture Overview

The plugin is divided into the four different python submodules depicted in Figure 7.3.

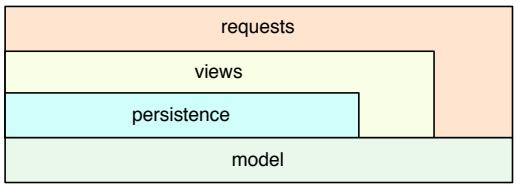


Figure 7.3: Main modules of the Adaptive Software Artifacts Plugin for Trac. Each module depends on the ones directly below it.

Model

The `model` module represents the core of the plugin. It is based on the ADAPTIVE OBJECT-MODEL architectural pattern to cope with the evolution and flexibility requirements and it defines the meta-model that allows end-users to manipulate the domain entities of the system – i.e., the Adaptive Software Artifacts.

Figure 7.4 shows two of the more important classes of this module – `Entity` and `Instance` – their relation with the language's built-in classes – `type` and `object` – a few exemplary Adaptive Artifact types – `Task` and `UserStory` – and some of their instances.

The `Entity` and `Instance` classes implement the TYPE OBJECT pattern using LANGUAGE PIGGYBACKING (p. 191). This means that the `Task` and `UserStory` types of Adaptive Artifacts are created at runtime with the statute of *classes*, and that specific

[2] An index of available plugins can be found in the *trachacks* website, in the Web address `http://trac-hacks.org/wiki/HackIndex`.

[3] Python is a general-purpose, dynamic, garbage collected, programming language. It focuses on object-orientation but allows different programming styles to be used, namely, imperative and functional programming styles.

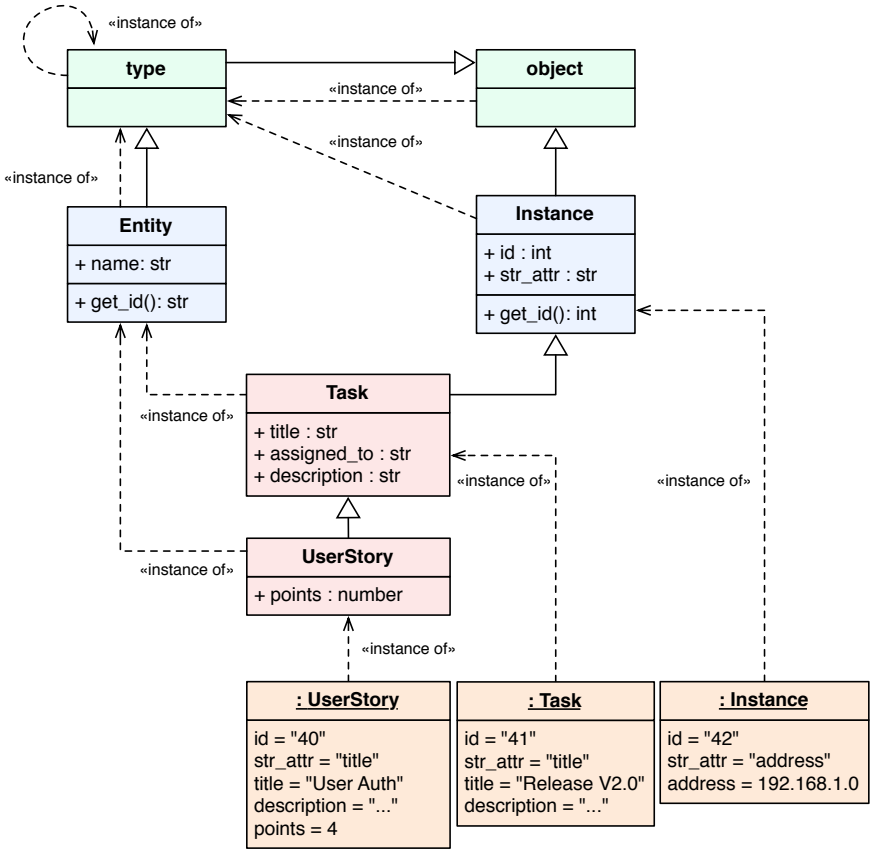


Figure 7.4: Instantiation and inheritance chains of the `Model` module depicted as a UML class diagram, with different colors to distinguish between different meta levels.

artifacts of those types are instances of those classes (shown on the bottom of the figure). `Entity` is therefore a *metaclass* and there are four different meta-levels at stake, represented by the four different colors in Figure 7.4.

Each new *type* of adaptive artifact may *inherit* from another *type* and specify its own attributes.

A note on terminology – the names `Entity` and `Instance` don't match the terms used to describe Type Object by Johnson and Woolf [JW97] – `TypeClass` and `Class` – or those used by Yoder et al [YBJ01] – `EntityType` and `Entity`. While this difference is mostly a matter of personal preference, it also has some historic reasons as the author previously contributed to an Adaptive-Object Model framework that used this terminology [FCA09]. Notwithstanding, in this context it could also be appropriate to use more specific names, like for example `AdaptiveSoftwareArtifactType` and `AdaptiveSoftwareArtifact`, but a secondary design goal advised against it: we attempted to keep the `model` module as generic as possible, so that it could be used in other projects that may benefit from using an Adaptive-Object Model.

Besides the `Entity` and `Instance` classes, this module also uses an `InstancePool` class. It keeps track of `Entities` and `Instances`, and provides the methods to query and access them. It can be seen in Figure 7.5.

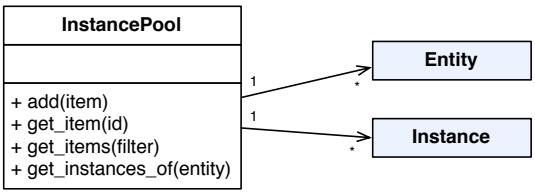


Figure 7.5: The `InstancePool`, `Entity` and `Instance` classes, depicted as a UML class diagram.

So far, `Task` and `UserStory` were shown using the UML notation for classes, but it is worth highlighting that they are instances of a metaclass, which makes them simultaneously classes *and* instances. In fact, while Figure 7.4 may allow to understand the design more easily, it doesn't make the use of the PROPERTY pattern apparent. Figure 7.6 depicts `UserStory` using the UML notation for objects. It shows the `__name__` and `__bases__` built-in python class attributes, and how the PROPERTY pattern is used to represent the attributes expected for each type of Adaptive Artifact – each attribute is an instance of an `Attribute` class, which allows to express not only the python identifier (`py_id`) of the attribute but also a `name` to be used for the user interface, a `domain`, and upper and lower `multiplicity` values.

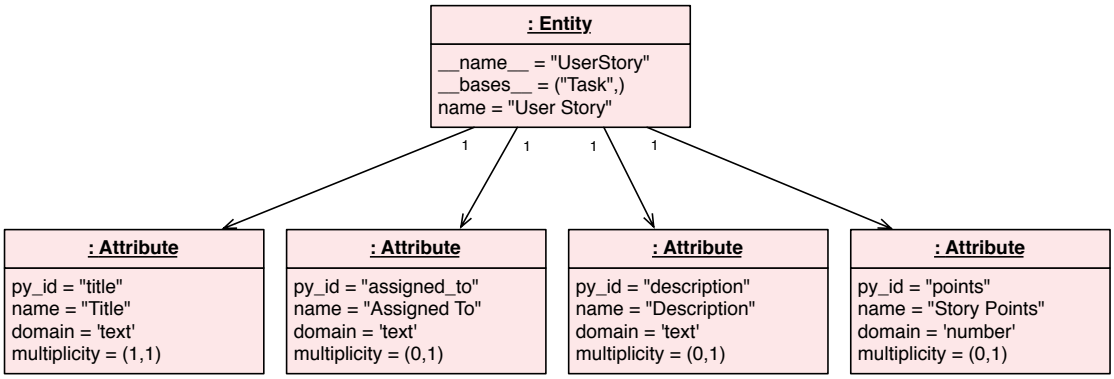


Figure 7.6: Example of an instance of the metaclass `Entity` named `UserStory` and its attributes, depicted as a UML object diagram.

Figure 7.7 shows the `Attribute` class together with the classes `Entity` and `Instance`. These three classes and the `Attribute` instances play the four roles defined by the TYPE SQUARE pattern.

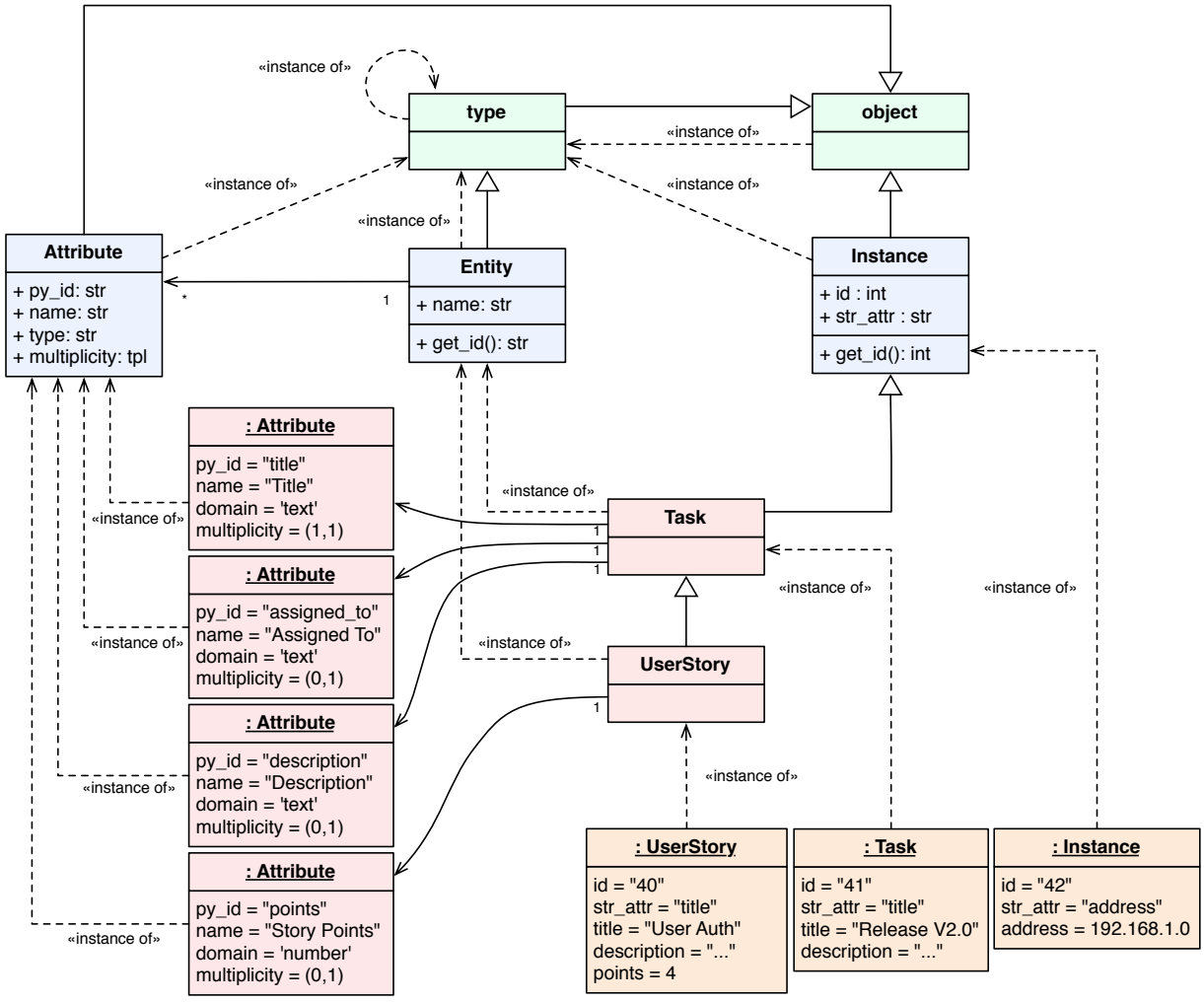


Figure 7.7: Instantiation and inheritance chains of the `Model` module, showing as instances the attributes of both of the classes that are `Entity` instances (`Vehicle` and `Car`). The notation used is that of a UML class and object diagrams.

Persistence

This module defines the database schema needed by the plugin and uses the `IEnvironmentSetupParticipant` extension point provided by Trac to create the required tables during installation[4]. Figure 7.8 shows the data tables used by the plugin to persist its data. The tables `asa_spec`, `asa_spec_attribute` and `asa_artifact` are used to persist respectively the instances of `Entity`, `Attribute` and `Instance`. The `asa_artifact_value` table is used to persist the attribute values of the instances of `Instance`.

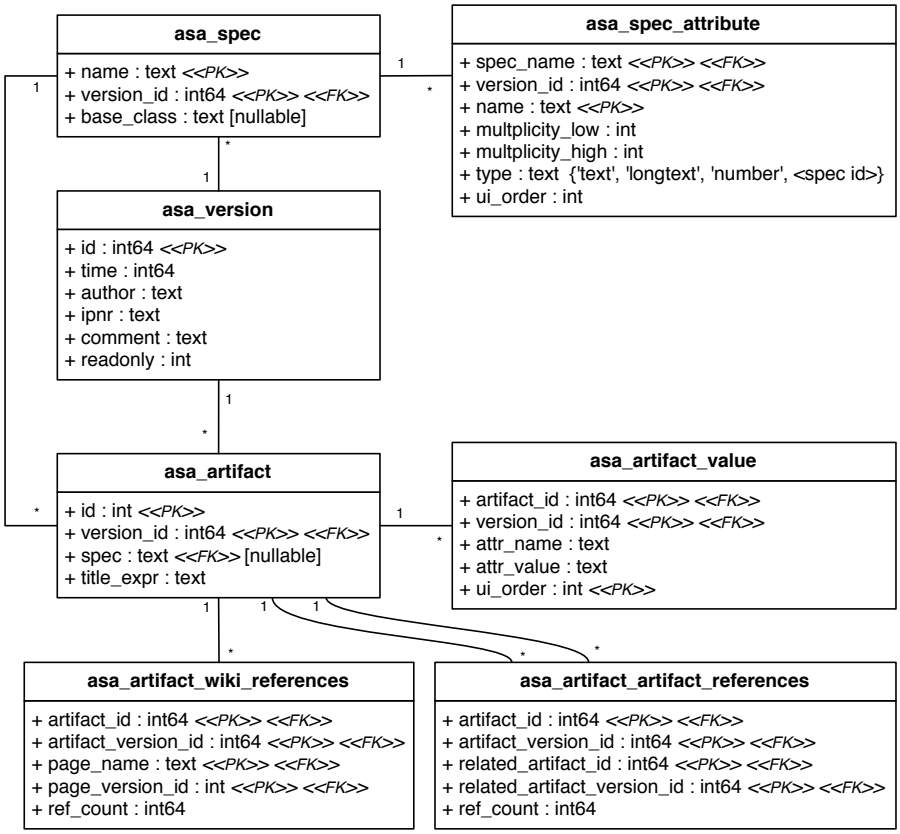


Figure 7.8: Database schema of the Adaptive Software Artifacts Plugin represented using a UML class diagram.

The database schema includes three additional data tables – `asa_version`, `asa_artifact_wiki_references` and `asa_artifact_artifact_references`.

The `asa_version` table keeps track of any changes made to the data. Changing data doesn't effectively *replace* the old values with the new, but creates a new *current*

[4] More details on the used extension points are provided in Section 7.4.2.

version of the relevant artifacts and/or artifact types. Even though the user interface of the plugin in the moment of this writing (i.e., version 0.5) doesn't yet allow to take full advantage of this version history, it is our goal to support the *observable* design principle as introduced in Sections 2.3.2 and 6.2.

Tables `asa_artifact_wiki_references` and `asa_artifact_artifact_references` track, respectively, the number of references made by each wiki page to each adaptive artifact, and the number of references made by each adaptive artifact to other adaptive artifacts. This data is used when presenting the index of adaptive artifacts, to provide useful contextual information and to support some of the Change Impact Awareness features (Section 6.3.5).

Additionally, this module provides two utility classes, a `Searcher` class, which encapsulates all text searches ran against the database, and a `DBPool` class, which encapsulates an `InstancePool` and provides the methods to load it with data on request. Each `InstancePool` is managed by a `DBPool`.

Requests and Views

The `requests` module provides a single `Request` class, which encapsulates HTTP requests to be handled by the plugin and routes them to the appropriate view. *Views* are represented by methods of the `views` module. Each Web-page provided by the plugin is handled by a specific view method that receives an instance of `Request` and returns an HTML template and the data structure to populate it. To build this data structure, view methods typically query the `persistence` and `model` modules.

7.4.2 Extension Points Used

The following ten Trac extension points are used by the plugin for purposes ranging from providing new items to the main menu to adding behavior to the wiki component, to data persistence.

`trac.env.IEnvironmentSetupParticipant` – Used by the plugin to provide its own data models, and create the associated database schema during the plugins' installation, or update the database schema and data when the plugin is updated to a new version.

`trac.resource.IResourceManager` – Used to make available within trac a new sort of software artifact (denoted as *resource* by trac) – Adaptive Software Artifacts. This extension point allows to assign control to the plugin over a particular base URL under which the new *resources* are made available.

`trac.search.api.ISearchSource` – Used by the plugin to make adaptive artifacts available within Trac's search mechanism.

`trac.web.api.IRequestFilter` – Allows the plugin to run logic before or after an http request to Trac. It is used, in particular, to make additional user interface behavior available within other Trac modules, like the wiki and the ticket system.

`trac.web.api.IRequestHandler` – Used by the plugin to manage the routing of its URLs.

`trac.web.chrome.INavigationContributor` – Allows the plugin to add its own item to the navigation menu.

`trac.web.chrome.ITemplateProvider` – Used by the plugin to provide its own templates and static resources, like icons, cascading style sheets and javascript libraries.

`trac.wiki.api.IWikiChangeListener` – Used to keep track of changes to wiki pages, and the references between them and adaptive artifacts resources.

`trac.wiki.api.IWikiMacroProvider` – Used by the plugin to provide a new wiki-macro that allows users to embed adaptive software artifacts in wiki pages.

`trac.wiki.api.IWikiSyntaxProvider` – Allows the plugin to provide a custom syntax rule to the wiki formatting system, which allows to link to adaptive software artifacts.

7.4.3 Features Walkthrough

Some requirements implied flexibility in the expression of structured contents, entailing challenges in the design of the user interface and interaction. Namely, many user interface elements needed to adapt to the different artifact types and artifacts, like is common in Adaptive Object-Model implementations [WYW07, FCA09].

The plugin makes a new option available on Trac's main menu that allows to access the index of adaptive artifacts. Figure 7.9 shows Trac with the option in question.

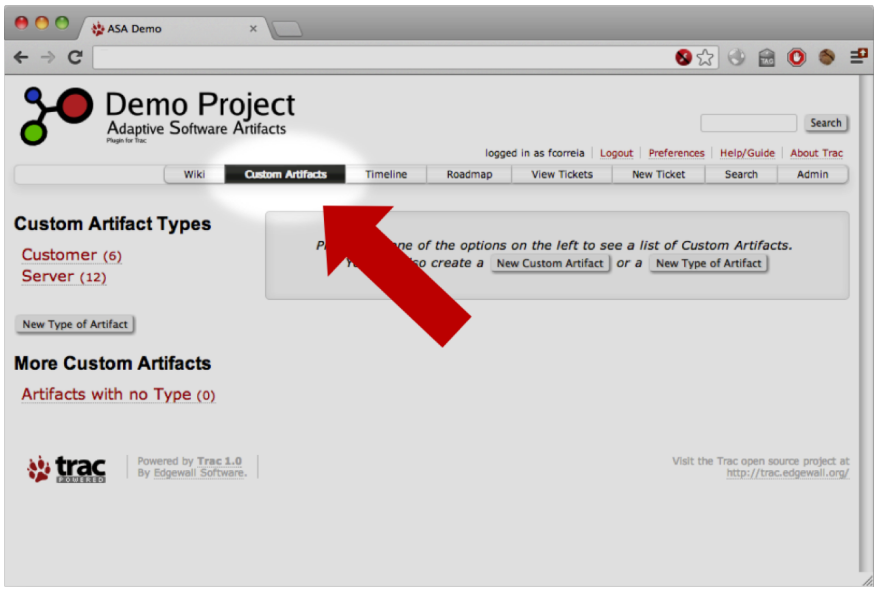


Figure 7.9: The option on Trac's main menu which allows to access the index provided by the Adaptive Software Artifacts Plugin.

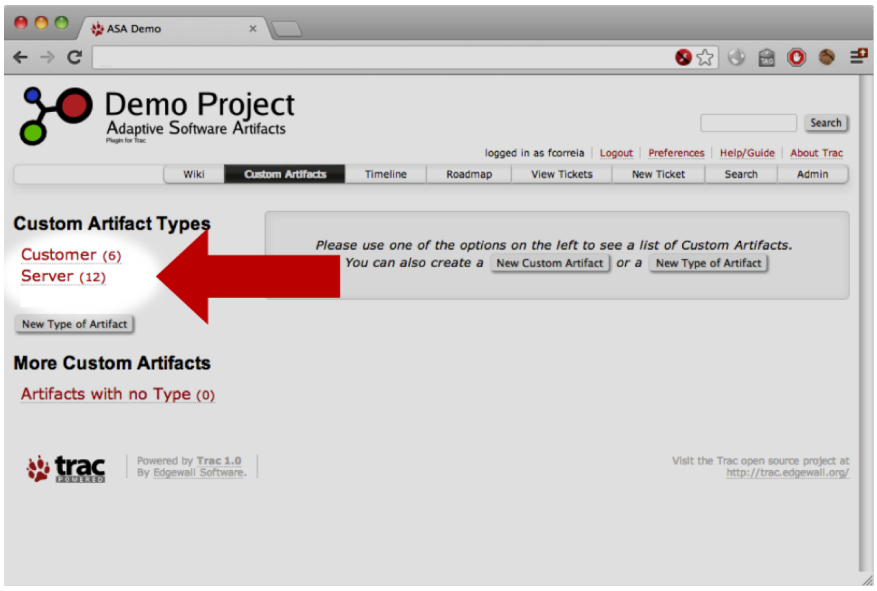


Figure 7.10: List of different types of adaptive software artifacts created by the developers.

Figure 7.10 highlights two types of adaptive artifacts that were already created, with the goal of helping the development team record various data about their *customers* and about the many *servers* where the team deploys their software. This example will be used in this section to illustrate the features and usage of the plugin.

In addition to the two artifact types, on the left side of the Web-pages is also visible a trivial example of *Smart Aggregation* (A3, p. 125) – an option to list all untyped adaptive artifacts. Figure 7.11 shows a list of concrete adaptive artifacts of the type

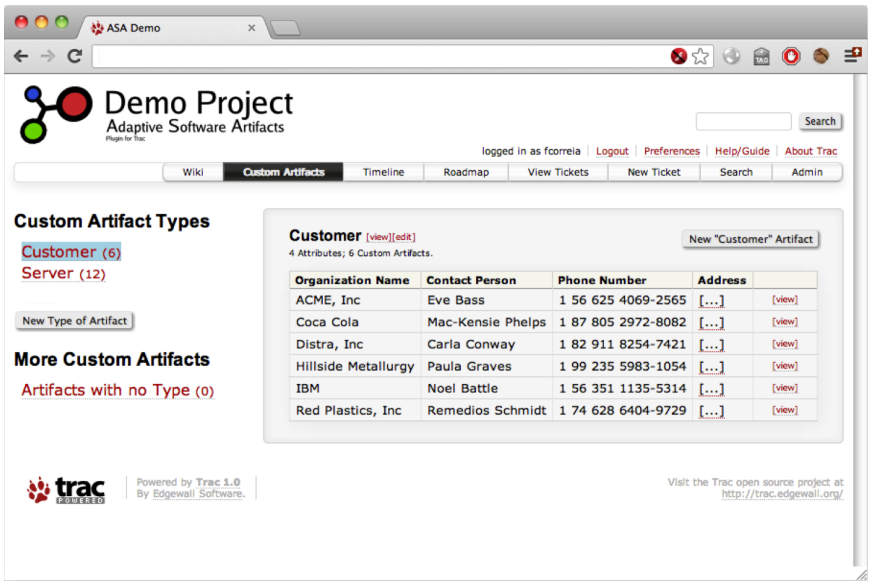


Figure 7.11: List of adaptive software artifacts of the type *customer*. All the attributes present in the listed artifacts are shown as columns, regardless if they are specified by the artifacts' type or not. Long and multi-line values are truncated.

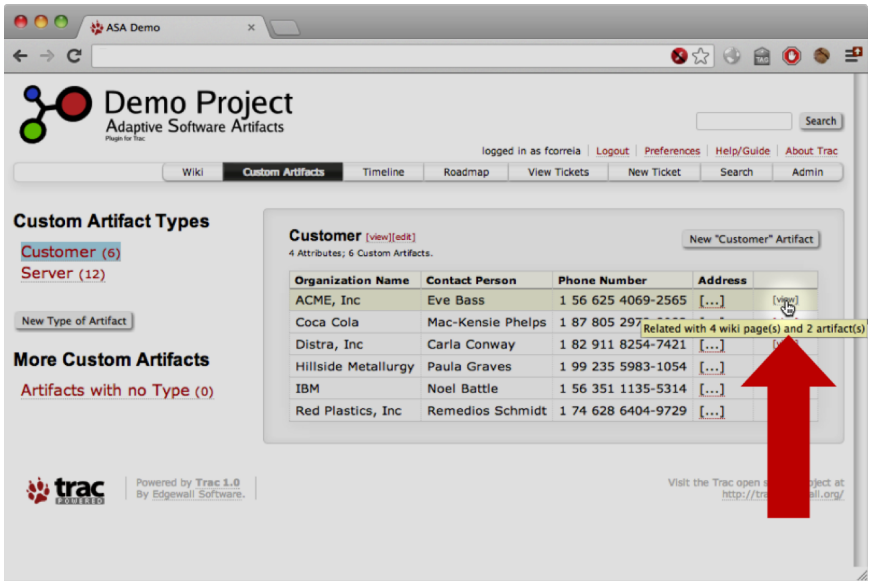


Figure 7.12: The *view* option, which allows to see the full details of any of the listed adaptive software artifacts.

customer. The attributes of each of them are visible in the list and more details are provided by the *view* option on each row (Figures 7.12 and 7.13). Besides the attributes of the artifact, the details page of an adaptive software artifact also lists all the wiki pages that refer to it, as well as other adaptive artifacts that refer to it too (Figure 7.13).

The sequence of Figures 7.9 to 7.13 illustrates *Navigation* (A4, p. 125), as they show how existing adaptive artifacts can be browsed to reach the wiki pages about a topic.

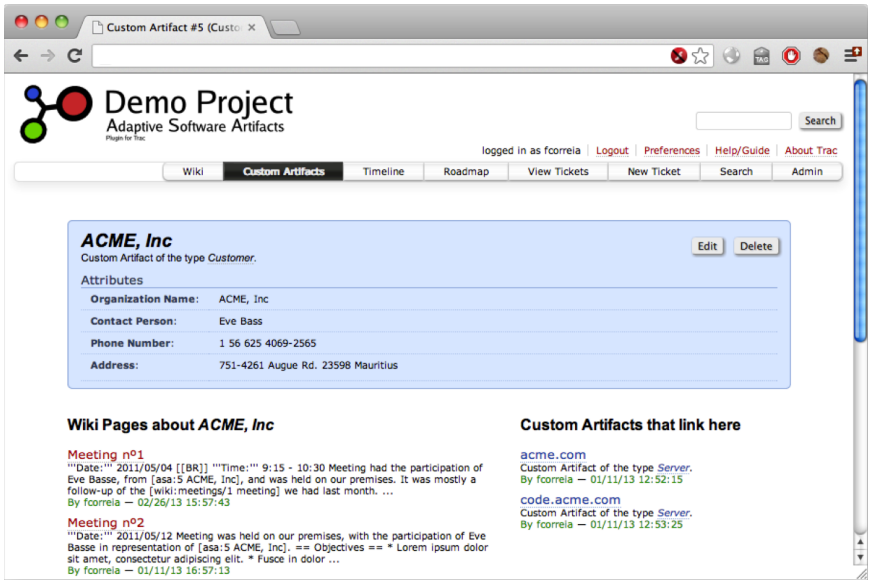


Figure 7.13: The details web-page for a specific software artifact, showing its attributes, the list of wiki pages that uses the artifact, and the list of artifacts that link to it.

Users can also easily *create new software artifact types* (A1, p. 123). Figure 7.14 shows

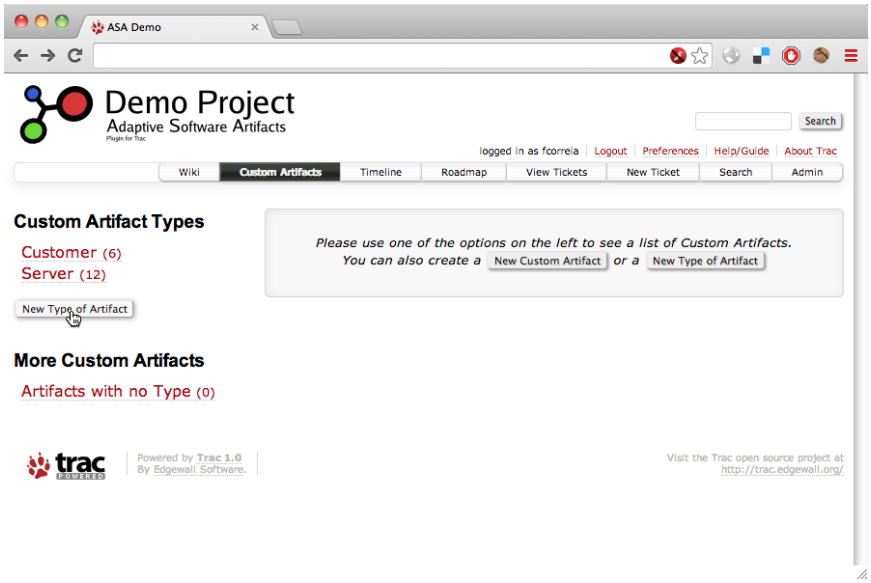


Figure 7.14: Button that allows to create new types of adaptive software artifact.

the mouse pointer over the button to do so from the index page. Figure 7.15 shows the page where that button leads, with the form fields required to specify the new type. This example shows the creation of an *Intervention* type, to record all the server interventions performed by the team (e.g., from hardware updates to the installation of new versions of the software).

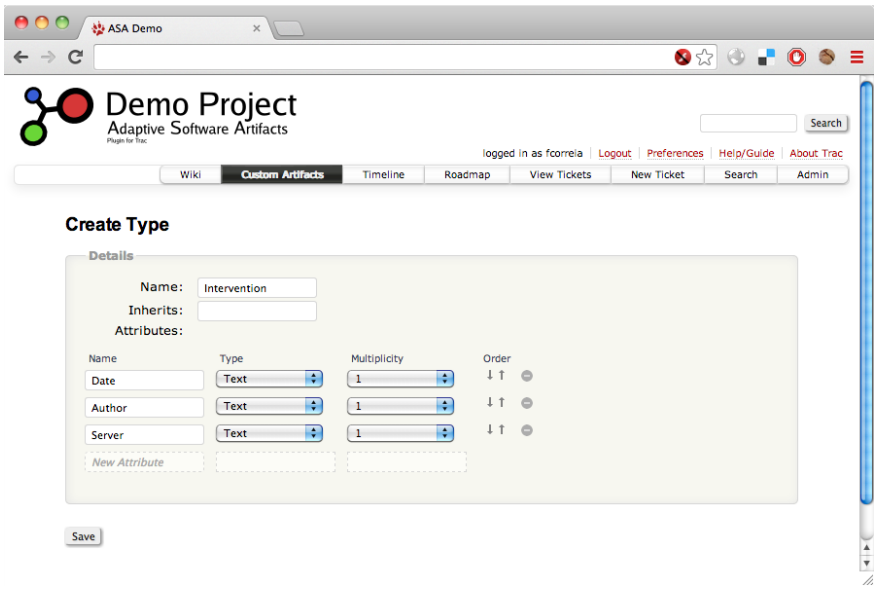


Figure 7.15: Creating a new type of artifact by specifying its name, the name of a super-artifact that it may inherit from, the artifact's attributes, the domain and multiplicity of each attribute and the order in which they should be presented to the user.

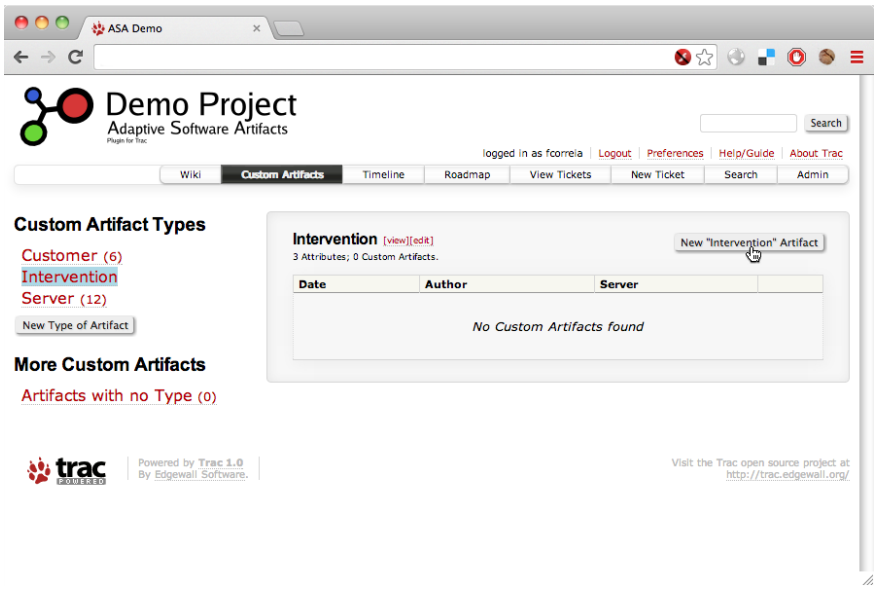


Figure 7.16: The empty list of artifacts of the newly created artifact type, and the button to create new artifacts of that type.

By selecting the new artifact type, one is then able to instance it as needed using the button indicated in Figure 7.16. Figure 7.17 shows the page and form used to

create a new artifact of this type. The attributes defined by the type are *recommended by the plugin, but not enforced* (A9, p. 127), which means that users are free to strictly follow those suggestions or simply add different attributes. The *title* selection allows to choose the attribute used to represent the artifact in links and listings (analogous to `toString()` methods available in programming languages like Java and C#).

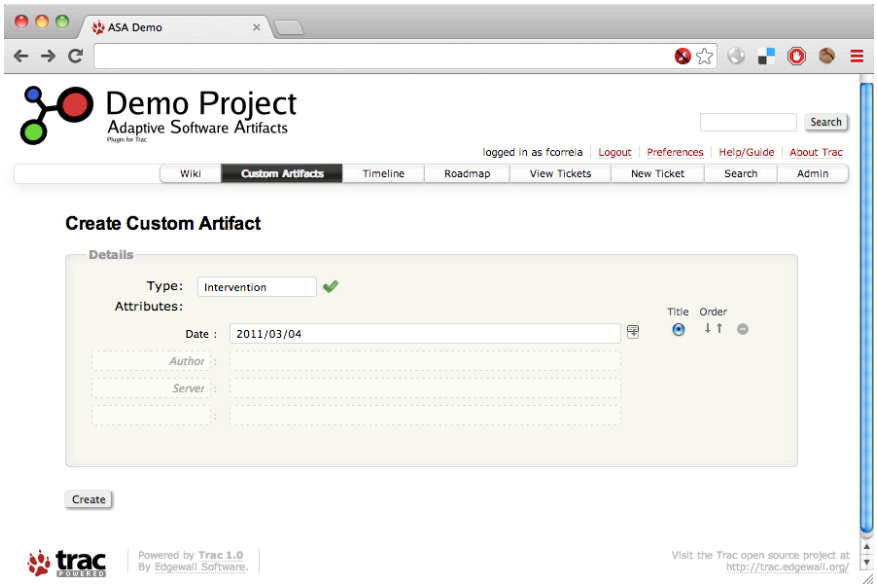


Figure 7.17: Creation of a new artifact of the type *Intervention*.

This approach to creating new adaptive artifacts and adaptive artifact types is in-sync with the *top-down* process to structuring information, described in Section 6.1, as users first define the artifact types and go on to instance them as needed. But one of

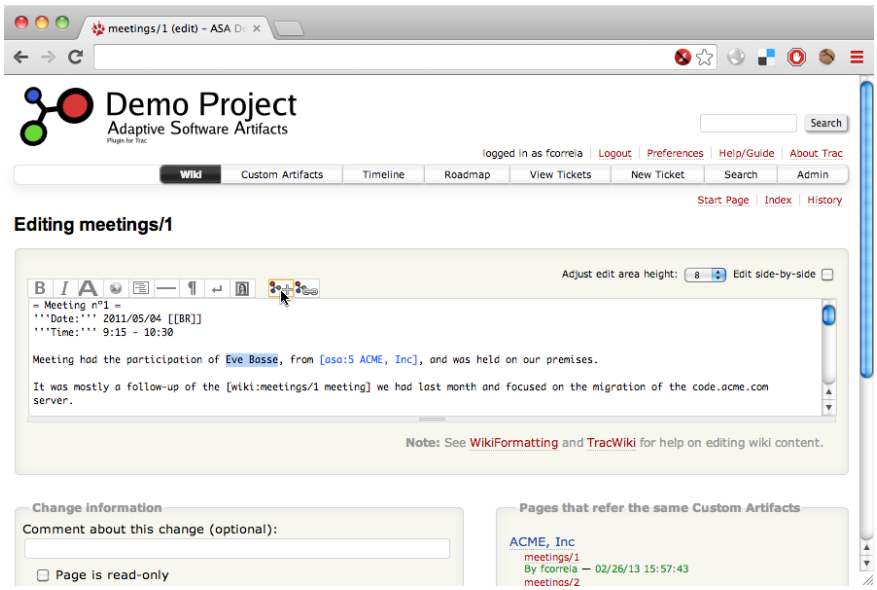


Figure 7.18: Text selection and the button to create a new adaptive artifact from that selection.

the goals of the plugin is also to support a *bottom-up* process, in which artifacts are created first, and types are derived only later, when it becomes apparent the existence of a set of common attributes. This process often starts in the context of free-form contents, hence the support provided by the plugin for *creating artifacts from textual contents* (A2, p. 124). Figure 7.18 shows the first step in this process, with the selection of a text fragment of a wiki page and the use of the indicated button to create a new adaptive artifact from that fragment. Figure 7.19 shows the modal dialog with which the user is presented to create the new artifact. The form in this dialog is similar in all respects to the one in Figure 7.17 with the difference that this dialog allows users to create new artifacts from within the wiki, without loosing context of the wiki page that is being edited. Upon creating the new artifact, the wiki markup for referencing it is automatically added to the selected text fragment, like can be seen in Figure 7.20.

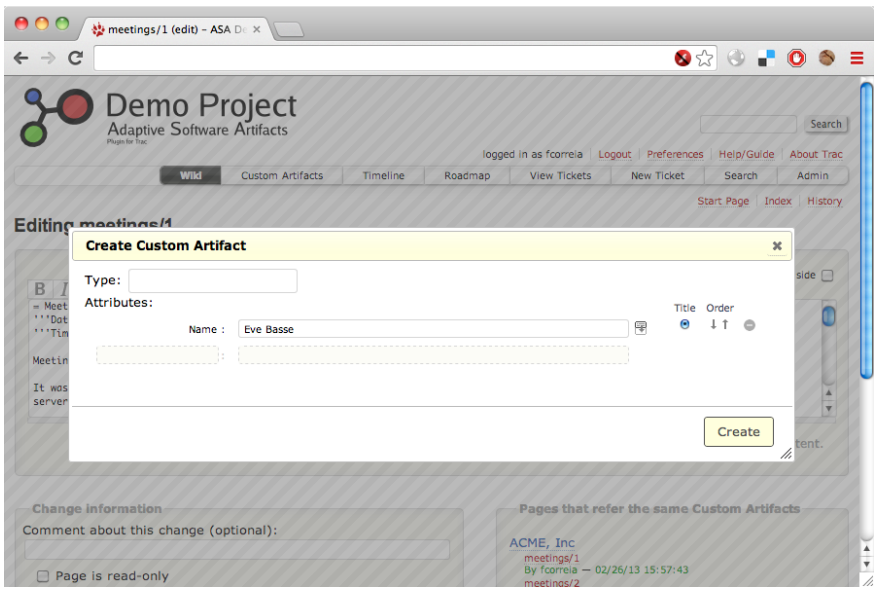


Figure 7.19: Form to create a new adaptive artifact from a text selection without loosing the context the current wiki page.

The *recommendation of connections from wiki pages to existing artifacts* (A7, p. 127) is illustrated by Figure 7.20. The plugin makes recommendations by underlining relevant text fragments with a dashed line. Overing these text fragments with the mouse pointer provides a contextual menu option to link to the artifact. This option shows the dialog on Figure 7.21, which allows the user to select the adaptive artifact that he would like to link to. Upon making that choice, the wiki markup to link to an artifact is added to the text fragment (Figure 7.22). This operation is similar to choosing to link to an existing artifact from an arbitrary text selection, which the user can also do through the toolbar *link* button.

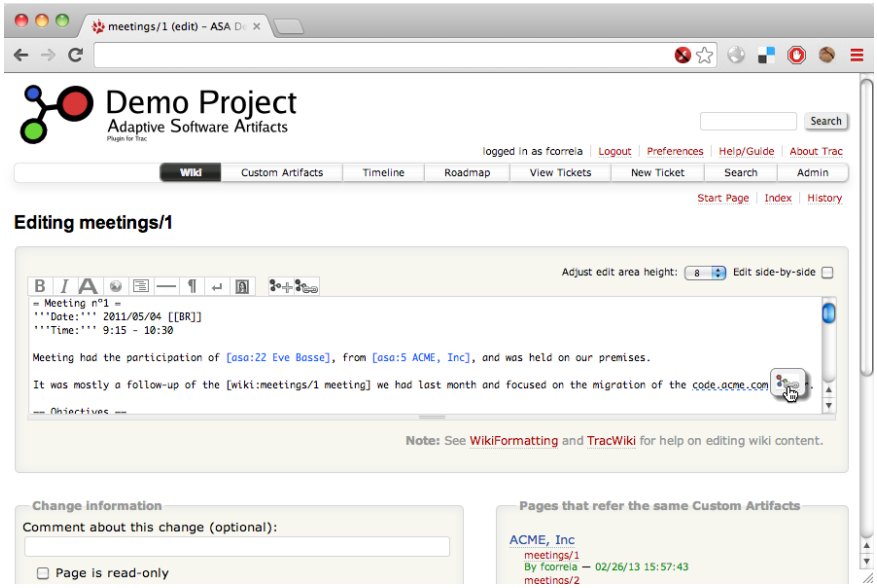


Figure 7.20: Recommendation of connection to an existing adaptive software artifact.

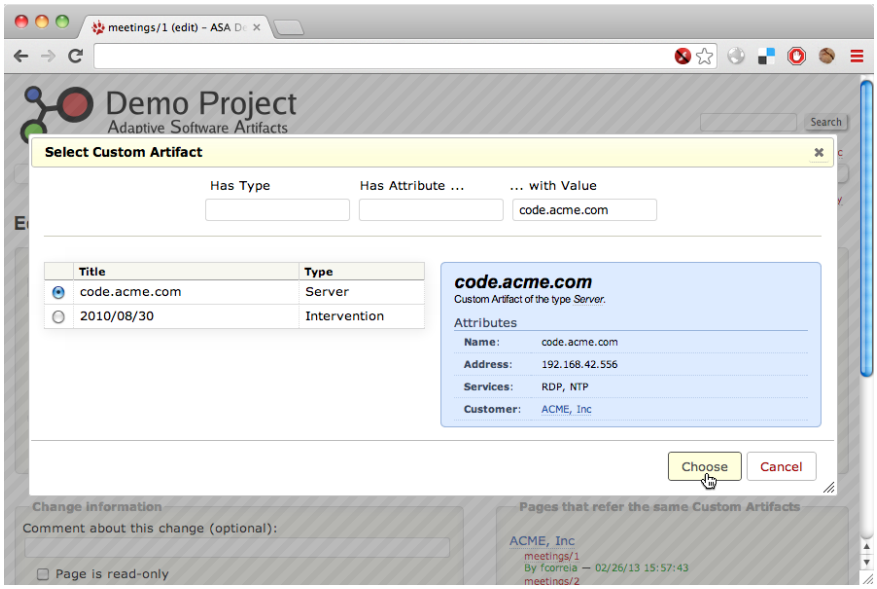


Figure 7.21: Choice of an adaptive artifact to link to from a text fragment in a wiki.

Figure 7.22 also shows the support provided by the plugin to help *maintaining consistency of the wiki contents* (A11, p. 127). When editing a wiki page a list of related wiki pages is shown on the lower right side. This list is built from the adaptive artifacts that the page has in common with related pages. Its goal is to bring to the user's attention the pages that might need an update to keep consistent with the current one.

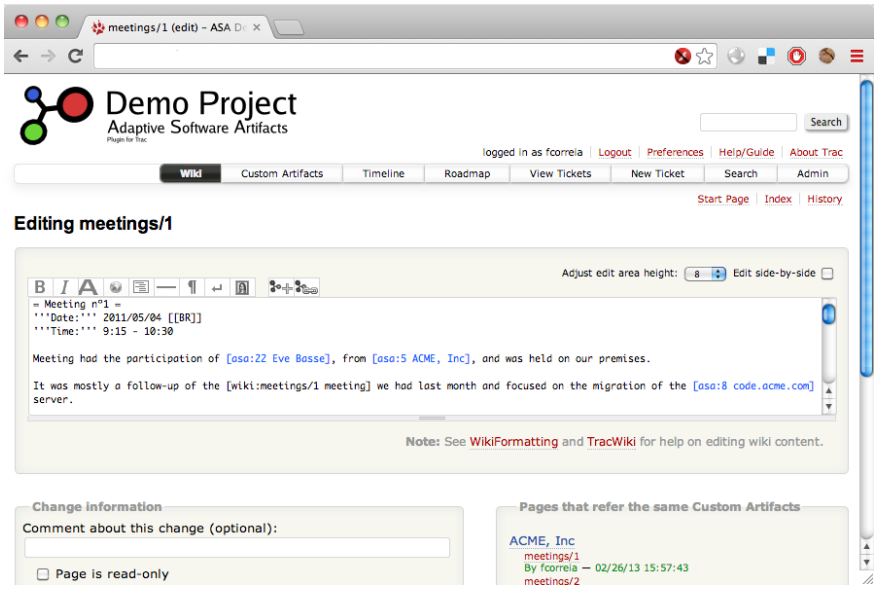


Figure 7.22: Wiki page after following a recommendation of the plugin to turn a text fragment into a reference to an adaptive software artifact.

Figure 7.23 illustrates the experimental feature introduced in Section 7.2, which tries to support a quicker understanding of the structured contents by representing

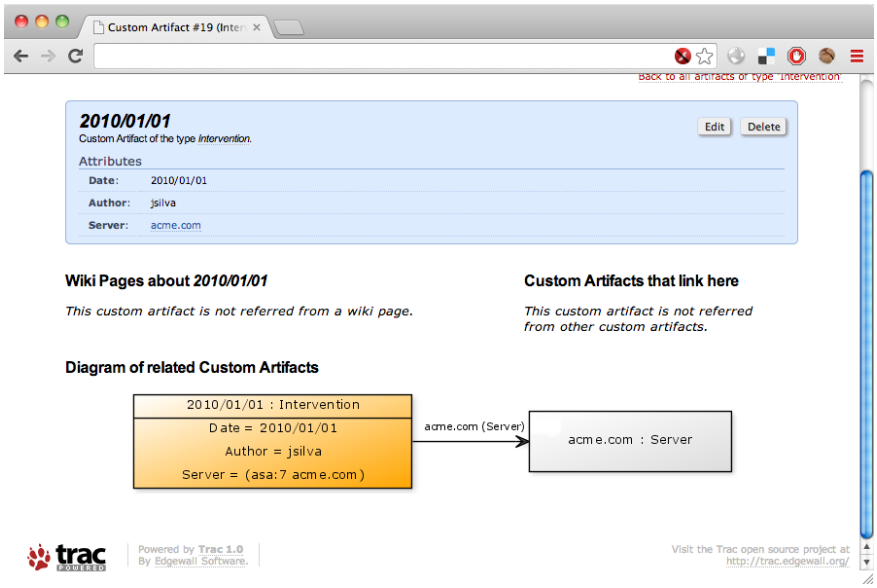


Figure 7.23: Showing an adaptive artifact's object graph as a UML object diagram.

an Adaptive Artifact and its relation with other Adaptive Artifacts as a UML object diagram. If the feature is activated in the plugin's configuration file, such a diagram is shown in the pages with the details of an Adaptive Artifact.

7.5 Development History and Analysis

The implementation of the plugin took place mainly between January 2011 and April 2013. The development saw different levels of activity during this period as presented in Figure 7.24.

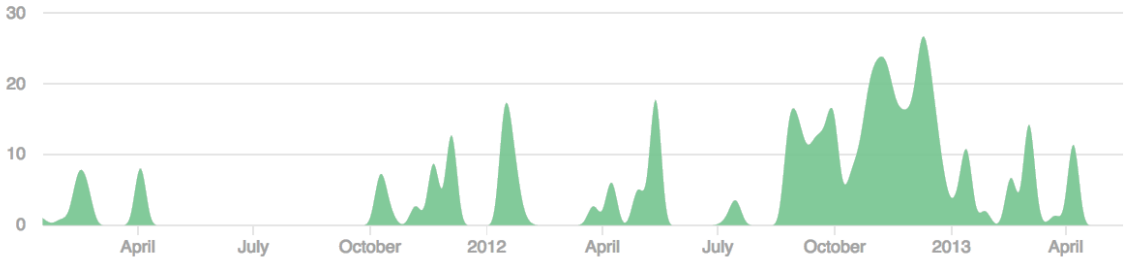


Figure 7.24: Timeframe and activity. Number of commits on the project over time since its creation in January 2011.

The codebase consists primarily of Python, as show in Table 7.3. Table 7.4 shows the number of source code statements excluding unit-tests and the test coverage of each module. Even though the unit-tests address all the python modules to some degree, they focus mostly on the `model` module, which we have considered to be the most critical – any defect in this module may have potentially big effects on the rest of the plugin.

Language	Code Lines	Total Lines	Percentage
Python	2354	3232	52.7%
JavaScript	1193	1531	25.0%
HTML	650	704	11.5%
CSS	486	591	9.6%
XML	60	76	1.2%
Total	4752	6134	100%

Table 7.3: Breakdown of the number of source code lines by programming language used in the Adaptive Software Artifacts Plugin.

	# Statements	# Covered	% Covered
`AdaptiveArtifacts`	216	66	31%
`AdaptiveArtifacts.model`	324	277	85%
`AdaptiveArtifacts.persistence`	494	275	56%
`AdaptiveArtifacts.requests`	63	14	22%
`AdaptiveArtifacts.views`	506	46	9%
Total	1603	678	42%

Table 7.4: Breakdown of the number of python statements (i.e., excluding comments and blank lines) and unit-test coverage by module.

7.6 Availability

The plugin is available as open-source, under the same License as Trac – the modified BSD license. The current version has *alpha* status and may not be suitable for all contexts; we encourage and support teams to try it and provide feedback conducive to new improvements. The source code and some documentation about setting up a development environment and installing the plugin is available on the Web address `https://github.com/filipefigcorreia/TracAdaptiveSoftwareArtifacts`.

7.7 Evolution Plans

Even though some features were left outside the scope of this work, the development of the plugin can be taken further to support all the features described in Sections 6.3 and 7.2 and it would be interesting to validate them in the context of further user-studies.

The existing code-base could also benefit from refactoring on specific areas. These could make the plugin even more modular and more focused, and ease the addition of new features. For example, the purpose of feature F14 introduced in Section 7.2 is purely to collect usage data useful in the context of the experiment described in Chapter 8, and it would make sense to extract that logic to an independent tracking plugin, which could prove useful in itself in other contexts.

Trac already provides a comprehensive text search feature, that is able to retrieve any of the software artifacts supported by the platform *out of the box*. This feature was extended by the plugin to retrieve adaptive software artifacts' contents, so that they could also be found through the same mechanism. Future development will make the feature more powerful by taking advantage of the automatic index (Sections 6.1

and 6.2), in line with the direction of research in recent years on the topic of semantic search [Mäo5].

7.8 Summary

The architecture and implementation described in this chapter is for a plugin for the Trac software forge that uses the Adaptive Software Artifacts approach detailed in Chapter 6. The feedback collected during the development was used to polish some aspects of the approach. In this sense, the plugin helped to validate the practicability of the approach and of implementing it in an integrated environment.

Rather than trying to encompass the whole approach, the development of the plugin has focused first on the core activities of the approach and especially on those of value to the experiment described in Chapter 8. The plugin played a vital part in this experiment, which provided new insights and helped to further validate the research.

The plugin was developed using mainly the python and javascript programming languages, and relies on Trac's component architecture the provided extension points to add new functionality to the platform. It's made available as open source software.

Chapter 8

Statistical Experiment

This chapter describes an experiment conducted with the goal of evaluating some properties of software documentation built using the approach introduced in Chapter 6. The Adaptive Software Artifacts Plugin detailed in Chapter 7 was used by a group of MSc students, who were asked to perform a series of programming tasks in a controlled environment. The design of the experiment considers two treatments – the *control* treatment, using a regular Trac software forge, and the *experimental* treatment, using a Trac software forge enabled with the Adaptive Software Artifacts Plugin. The data collected during the experiment was to a great extent numeric and was subject to a statistical analysis. The following sections present the full design of the experiment and its results.

8.1 Goals

The goal of the experiment is to focus on knowledge acquisition, thus covering four of the issues introduced in Section 4.5. Specifically, it tries to provide an answer to:

I1. Efficiency of Knowledge Acquisition.

Do developers spend less *time* acquiring knowledge from the contents?

I1.1. Efficiency of acquiring information structure.

Do developers spend less *time* understanding the contents?

I1.2. Consistency of the contents.

Are resulting contents more *consistent*?

I1.3. Quality of the classification scheme.

Do developers spend less *time* searching for the contents they need?

A secondary goal of this first experiment fits within the broader research around Adaptive Software Artifacts. We hope that the results of this study will encourage software companies to welcome case-studies with the plugin in their contexts. Two software companies expressed interest in the Adaptive Software Artifacts approach, and we believe they may be interested in trying the plugin if they see promise in the results of this experiment.

8.2 Design

Has highlighted by Dean and Voss [DV99], experiments rely on three basic techniques – *replication*, *blocking*, and *randomization*. The contents of this section describe the result of using these techniques to design the experiment and all that was taken into account in that process.

8.2.1 Participants

The recruitment was done among students and tried to balance the *research value* with the *pedagogical value* that empirical studies in an academic setting should provide [CJMS10]. Students were invited by e-mail to participate in the study and those interested signed-up using an on-line form. This was an opportunity for them to apply directly some of the knowledge obtained in one of their courses, so they were offered a small bonus on the course's final grade for participating of the experiment. This was the main incentive that the subjects were given and was enough to attract some participants.

It's important to note that even though some of the options mentioned in this experimental design come from the use of students as subjects, we believe the same design can be easily used to replicate the experiment in other contexts.

The subjects were randomly assigned to the treatments[1] but there was still a concern that some of them could possess considerably different skills that could lead to statistical deviations of the results. To ensure that this was not the case, their background was evaluated by considering their grades on a set of relevant courses and by considering their answers to a background assessment questionnaire. The specific details of how these analyses were made are described in more detail in Sections 8.3 and 8.4.1.

[1] The assignment was done with the assistance of *ASAAnalizer*, a software module that will be further detailed in Section 8.2.8

8.2.2 Data Sources

Five main sources of data were used for the experiment: a) the university's information system, where we obtained information about some of the subjects' grades, b) the usage logs of the Trac environments, c) the tasks' completion times, recorded by the participants, d) the source code that was produced, and d) the questionnaires answered before and after the treatments. Section 8.2.5 will go into the details of how the data was collected from each of these data-sources.

8.2.3 Environment

The experiment was ran in one of the University of Porto's computer laboratories, which provided a setting familiar to the participants and allowed to control conditions of the software environment. Each participant had available **a workstation**, pre-installed with the Eclipse IDE and the Java Development Kit, and the **Trac environment**, with or without the Adaptive Software Artifacts Plugin, depending on the treatment group. All Trac environments were pre-populated with documentation about the software system to be developed, namely, about the problem domain, about a framework to be used and about the sequence of tasks to perform.

8.2.4 Procedure

The participants were equally distributed in two groups, corresponding to the two different treatments – the control group (CG) used a regular Trac environment, and the experimental group (EG) used a Trac environment with the Adaptive Software Artifacts Plugin.

The entire session took around 1h45 for each group and can be summarized as the following steps, the relative durations of which are illustrated in Figure 8.1.

01. **Introduction** (10 min)
Participants were explained the overall context and high-level goal of the experiment, but not how the conditions of their group differed from the ones of the other group. They also received a short presentation on how to use Trac's basic features, with those on the experimental group also receiving an explanation of the features provided by the Adaptive Software Artifacts Plugin.

02. **Background Questionnaire** (5 min)
Before starting the tasks themselves, the participants were asked to answer a short background

questionnaire, so that later it could be confirmed if there wasn't any statistic deviation of their skills or experience that could influence the results of the two groups.

03. **Programming Tasks** (80 min)
The participants were given an envelope with instructions; explaining briefly the software environment that they had available, how they could login to the Trac environment and what was expected from them in general terms. The eclipse environment was pre-configured on each workstation to avoid any setup issues. The Trac instance provided all the documentation, including some technical documentation, some domain documentation, and the description of the tasks themselves.

04. **Assessment Questionnaire** (10 min)
The participants answered a set of questions designed to assess some effects that cannot be measured objectively by the environment. These questions are mostly related with cognitive issues and are described in more detail in Section 8.2.5.

Figure 8.1: Experiment steps and their relative durations.

The full materials used in the experiment can be found on Appendix B, including the questionnaires, the instructions given to the participants and the description of the tasks provided in the Trac environment.

8.2.5 Data Collection

The student grades, for the selected set of courses, were manually collected from the university's information system into a digital spreadsheet. To complement this data we have also asked participants to answer a background questionnaire before the experiment itself started. The questionnaire was designed with 15 five-level Likert items [Lik32] with the format: (1) *strongly disagree*, (2) *somewhat disagree*, (3) *neither agree nor disagree*, (4) *somewhat agree*, and (5) *strongly agree*. It was provided in paper form and the answers were later transcribed manually to a digital spreadsheet. The background questionnaire items are reproduced in Appendix B.

The Trac environments used in the experiment were instrumented to log detailed information about their users' activities. More specifically, an ancillary functionality

of the plugin is to collect the amount of time spent on different areas of the Trac environments, distinguishing how much of it was spent on wiki pages, on using Trac's search feature, on viewing adaptive artifacts, or on accessing the adaptive artifacts' index.

Collecting the time taken to complete each task required user input and was thus left outside the scope of the plugin; it was part of the written instructions that the participants should record these times on paper. Appendix B also reproduces the instructions sheet that was provided to the participants and, in particular, the table where they were asked to record the tasks' start and completion times.

Both the activity times and task times can be used to address I1, I1.1 and I1.3, three of the issues highlighted in Section 8.1:

> **I1. Efficiency of Knowledge Acquisition.**
>
> Do developers spend less *time* acquiring knowledge from the contents?
>
> > **I1.1. Efficiency of acquiring information structure.**
> >
> > Do developers spend less *time* understanding the contents?
> >
> > **I1.3. Quality of the classification scheme.**
> >
> > Do developers spend less *time* searching for the contents they need?

These forms of data collection allow to capture a few time-related attributes, but a lot of the attributes of interest result from human cognitive processes and don't produce external, easily measurable, outcomes. For this reason, participants were also asked to answer an assessment questionnaire in the end of the experiment. The questionnaire was designed using the same five-level scale as the background questionnaire and was comprised by 20 Likert items. The items were organized into 6 different categories.

> **External Factors.** These items help removing validation threats concerning some of the experimental conditions, like the physical environment and software operational issues.
>
> **Overall Perception.** The items of this category provide a high-level view of how subjects felt about the platform and the tasks.
>
> **Information Attributes.** Supports understanding how the subjects saw the information that they had available.
>
> **Classification.** Helps assessing how the subjects saw and used the platform's features to their advantage, in particular, those related to finding the contents they needed.

Understandability. These questionnaire items allowed to gain insights on how easy it was for subjects to understand the information provided by the platform.

Consistency. Supports understanding the subjects' perception regarding the consistency of the contents that they were provided with.

Additionally, three final open-ended questions were included to help detecting any non-conformance with the outlined process, identify alternate explanations that may have not been considered before, or even to help recognizing additional threats to validity [CJMS10]. The assessment questionnaire is also reproduced in Appendix B.

The answers to the questionnaire items are reproduced in full in Appendix C and were used to address all the issues previously highlighted in Section 8.1:

I1. Efficiency of Knowledge Acquisition.

Do developers spend less *time* acquiring knowledge from the contents?

I1.1. Efficiency of acquiring information structure.

Do developers spend less *time* understanding the contents?

I1.2. Consistency of the contents.

Are resulting contents more *consistent*?

I1.3. Quality of the classification scheme.

Do developers spend less *time* searching for the contents they need?

The produced source code was also used as a source of data. In the end of the experiment the participants were instructed to create a zip file of their source code and to attach it to a specific page of the Trac environment. Later, we have looked at the source code directly to confirm the completeness of the tasks. The result of this analysis didn't contribute directly to addressing any of the issues that we set out to answer, but served as a quality control for the task times reported by the subjects.

8.2.6 Data Analysis

A software module was developed primarily for the purpose of supporting the data analysis. For analyzing the data collected during the experiment the module takes as input a) a spreadsheet with the **times of the tasks** as recorded by each participant, b) a spreadsheet defining the locations of the two Trac databases that contain the **activity data** for the control and experimental groups and c) a spreadsheet containing all the **answers to the questionnaires**. Running this module outputs a) **CSV files**

containing the result of aggregating the data, calculating descriptive statistics and running statistical tests like the *t*-test and the Mann-Whitney U test, b) some **PDF files** containing bar-charts and boxplots of some of the results and c) **TeX files** containing formatted document fragments that were included in this thesis.

The development of this module allowed systematizing the analysis process and make it more easily reproducible. Its broader role in the experiment and the part it is meant to play in its replication is described in Section 8.2.8 and more details about its high level design are included in Appendix D.

8.2.7 Pilot Experiments

Two pilots were used to fine-tune the experimental design. They were run with the goal of testing the protocol and identifying unsuspected problems in the instructions, in the plugin, or in the data collection [DV99]. Due to operational constraints, only one subject was used in each pilot. The pilots strictly followed the experimental protocol as it was defined at the time, and were run in the same environment and using the same setup that the main experiment would later use. Participants of the pilot were asked to *think aloud* during the execution of the instructions, to encourage a description of their perceptions and expectations towards the information that they were presented with and towards the actions that they had to perform [Nie93, p. 195]. This enabled us to identify and correct several issues in the experiment's design and in the clarity of the introductory presentation, the questionnaires and the written instructions.

8.2.8 Replication

Controlled experiments are not as used in software engineering research as in other fields, and those that are conducted often aren't independently replicated [SHH+05, CJBV13]. Yet, replicating controlled experiments is key to reach sound empirical evidences.

To support and encourage better independent replications and additional studies, researchers designing an experiment often provide an *experimental package* that can be used as a guide. Section 8.2 and Appendix B effectively constitute such a package, and we encourage other researchers to independently validate the results of this experiment[2]. It addresses and provides guidelines for the subject recruitment and

[2] These contents have also been made available publicly, on the Web address `http://softeng.fe.up.pt/esseWiki/doku.php?id=tenfogs03:start`.

assignment process, the environment requirements, the research procedure, the data collection mechanisms and the analysis techniques.

When creating experimental packages, researchers should systematize the research process, the experiment materials and the analysis and reporting of the results to the furthest extent possible [GBeA07]. But even when packages are carefully created it can still be a challenge to convey the complete know-how of an experimental design, as it usually still depends to some degree on tacit knowledge [SBC+02].

The *ASAAnalyzer* is a software module that we have developed in the context of this research with the goal of automating a part of the process of analyzing the collected data, and making that process repeatable. The module can be used in several phases of this experimental design, like the random assignment of subjects to the experimental groups, the analysis of the answers to the background questionnaire, the analysis of the data collected from the Trac environment and the analysis of the answers to the assessment questionnaire. It is made available as open-source software and Appendix D includes relevant high-level documentation on its features and operation[3].

8.2.9 Discussion

One of the concerns in the early design of the experiment was its scope. Should it focus more on some of the research issues? How reliably should it try to recreate real-world conditions? What measurements should be made? The answers to these questions depended as much on the object of study as on the available resources.

In practice, the process of knowledge *capture* is often closely intertwined with that of knowledge *acquisition*. Studying the two phenomena together could provide some insight on this relation but we anticipated that it would be very difficult to excerpt control over the amount of different variables that this would imply. In other words, it is easier to isolate cause-effect relationships by studying the two events separately. This should be emphasized as, even though this experiment focuses on knowledge acquisition issues, we consider knowledge capture issues to be no less important and should receive our attention in future user-studies.

The choice of the programming tasks was another concern. Our design had to be consistent with the subjects' time constraints and availability but we also wanted the subjects' software documentation needs during the experiment to be motivated by credible software development tasks and we believed that the issues that we wanted to

[3] It can also be found on the Web address `http://github.com/filipefigcorreia/asaanalyzer`, together with the source code.

tackle only pose themselves when non-trivial amounts of information are at stake. The tasks needed to be simple to implement but also complex enough that subjects would feel the need to refer to existing software documentation. This was the main reason why we resourced to an existing, well documented, software framework. JHotDraw is a well documented framework and revealed itself a more credible starting point than if the documentation would have been completely synthesized by the subjects or the researchers. Starting with a significant amount of documentation and source code ensured that knowledge of a meaningful amount and complexity was being made available. The tasks did not require the subjects to spend much time programming but required a knowledge of the framework that the subjects did not possess and needed to acquire.

Other design considerations were related with the data collection process. Some effects are difficult or impossible to measure directly, especially when they are part of human cognition. On the one hand, *measurements* are more objective and reliable, but they hardly reflect directly what we want to find out. On the other hand, questionnaire items are intrinsically subjective and depend to a great extent on what the subjects are able to remember, but the answers that they provide can be closer to the research questions. Both data collection methods may provide useful insights and they were combined to improve the knowledge of the researchers concerning the questions at hand.

8.3 Running the Experiment

The experimental design described in Section 8.2 covered the most important concerns, from the recruitment of subjects to the data collection and analysis. However, unanticipated conditions may arise from the specific context of how, where, when and who participate of the experiment. We believe this knowledge is important to put the experimental process and results into context, and may help other researchers trying to replicate the experiment.

8.3.1 Participation

The experiment gathered the participation of students from the Integrated Masters in Informatics and Computing Engineering, lectured at the University of Porto, College of Engineering. All were students of the 3^{rd} year and were enrolled on the course of *Object Oriented Programming Laboratory*. The invitation to participate of the user-study

was sent to the 130 students of the course, of which 61 registered for the experimental sessions. Due to other academic activities some of the students that had registered for the sessions were not able to effectively attend, and we counted with 43 subjects on the day of the user-study.

Besides the different Trac environments, the main difference between the two treatment groups was that only the Experimental Group was given an overview of the features provided by the Adaptive Software Artifacts Plugin. This was the main reason that lead us to schedule the two groups to two distinct time-slots. On the other hand, if we were to have all the subjects present simultaneously it could have allowed adapting more easily to the absence of some of the students, and re-assign them to the groups so that the number of participants could be distributed more evenly.

8.3.2 Data Quality

The quality of the data collected manually was one of the concerns. Of the 43 subjects that participated of the experiment, some had to leave earlier and did not complete the assignment that was given to them, nor answered the assessment questionnaire. We have also detected from the delivered source code that two of the subjects copied from each other, even though they were told that only their participation, and not the result of their work on the session, would have an effect on their grade. Through the analysis of the delivered source code we have also detected that some tasks were not completed in full even though their authors believed they were. These issues were dealt with case-by-case during data analysis.

8.4 Data Analysis

The data collected over the course of the experiment was mainly quantitative and was the focus of statistical analyses. Some of them were significance tests, the choice of which was based mostly on the robustness of the tests and on their assumptions. The Mann-Whitney U test is generally more robust and was made our default choice. It does make an assumption of equal variances though, so an alternative was chosen when dealing with data with unequal variances. Standard t-tests also make an equal-variance assumption, so we have resourced to the unequal-variance variant of the t-test (i.e., Welch's test) for such cases.

This section details the approaches used to examine the different kinds of data and the results of those analyses. The notation it uses denotes H_0 as the null hypotheses,

H_1 as the alternative hypotheses, CG and EG as the control and experimental groups, u as the u-statistic of Mann-Whitney U tests, t as the t-statistic of t-tests, and ρ as the probability of wrongly rejecting H_0. The significance of all statistical test results was interpreted using a 95% confidence level. The number of subjects of the CG and the EG was respectively 19 and 15.

8.4.1 Background

The subjects' background was evaluated by comparing their grades on a set of relevant courses, to ensure there were no significant differences between the two groups. The goal is to validate that any difference in the results was a product of the *treatments* and not due to any differences between the subjects in the two groups themselves.

The complete grades are included in Appendix C and are summarized in Table 8.1. We can observe that the means of the grades of the two groups are indeed very similar.

		Grades	
	# Subjects	$\bar{x}$	σ
Control Group	19	15.53	2.1880
Experimental Group	15	15.08	2.1798

Table 8.1: Descriptive statistics of the students' grades by treatment group.

It was then assured that an equal variance could be assumed (Appendix E) and an independent-samples Mann-Whitney U test was used to establish a comparison between the grades of the control group and those of the experimental group. Table 8.2 present the results of the test, showing that there was **no significant difference** in the grades of the control and experimental groups ($\rho > 0.05$).

H_1	u	ρ
$\neq$	167.000	0.404

Table 8.2: Result of a two-tailed independent samples' Mann-Whitney U test for the comparison of students' grades. The two-tailed ρ-value higher than 0.05 allow us to conclude that there is no statistically significant difference between the two groups.

A second evaluation of the subjects abilities focused on the data collected through the background questionnaires and thus in a more specific set of skills and knowledge. The results for each questionnaire item are analyzed next. Table 8.3 presents a summary

of the answers and the full data can be found as part of Appendix C. Supporting calculations and analysis can be found in Appendix E.

	CG			EG			MW-U		
	1 2 3 4 5	$\bar{x}$	σ	1 2 3 4 5	$\bar{x}$	σ	H_1	u	ρ
BG1.1		3.789	0.614		4.067	0.680	$\neq$	112.000	0.245
BG1.2		4.211	0.614		4.200	0.542	$\neq$	145.000	0.936
BG1.3		2.368	1.265		2.467	0.957	$\neq$	132.500	0.733
BG1.4		1.526	1.094		1.533	0.618	$\neq$	118.500	0.324
BG1.5		1.474	1.045		1.267	0.442	$\neq$	140.500	0.944
BG1.6		2.316	1.126		2.133	1.024	$\neq$	155.000	0.664
BG1.7		1.211	0.614		1.333	0.596	$\neq$	122.500	0.308
BG1.8		4.368	0.581		4.067	0.573	$\neq$	179.000	0.153
BG1.9		3.789	1.104		3.400	1.083	$\neq$	174.500	0.249
BG1.10		2.053	1.191		1.933	1.181	$\neq$	150.500	0.779
BG1.11		2.053	1.276		1.800	0.909	$\neq$	150.500	0.780
BG1.12		3.263	0.909		2.867	1.024	$\neq$	166.500	0.393
BG1.13		2.895	1.021		2.800	0.833	$\neq$	147.000	0.880
BG1.14		2.789	0.893		3.000	1.265	$\neq$	121.500	0.459
BG1.15		3.158	0.744		3.333	1.011	$\neq$	129.500	0.648

BG1.1. I have considerable experience using the Java programming language.
BG1.2. I have considerable experience using the Eclipse IDE.
BG1.3. I have considerable experience using Software Forges.
BG1.4. I have considerable experience using the Trac platform.
BG1.5. I have considerable experience using Trac's Adaptive Software Artifacts or Custom Software Artifacts.
BG1.6. I have considerable experience using frameworks.
BG1.7. I have considerable experience using the JHotDraw framework.
BG1.8. I have considerable experience with object-oriented software development.
BG1.9. I have considerable experience extending a system using composition and subclassing.
BG1.10. I have considerable experience developing industry-level applications.
BG1.11. I have considerable experience maintaining/modifying industry-level applications.
BG1.12. I have considerable experience documenting software systems.
BG1.13. I have considerable experience using technical documentation of software systems.
BG1.14. I have considerable experience using wikis.
BG1.15. I have considerable experience developing standalone GUI (Graphical User Interface) applications.

Table 8.3: Summary of the answers to the background questionnaire, including histograms and descriptive statistics for each question and the result of comparing the answers of the Control Group (CG) with those of the Experimental Group (EG) using a two-tailed Mann–Whitney U (MW-U) statistical test.

As seen in Table 8.3, let the alternative hypotheses (H_1) be CG $\neq$ EG for each background question – there was **no significant difference** between the scores of the control and experimental groups. This result corroborated the analysis of the grades and helped to reject the existence of differences between the two groups.

A few more observations can be made from this data. Namely, that it is the subjects' subjective opinion that they possess considerable experience using the Java programming language (BG1.1), using object-oriented development (BG1.8) and using

the Eclipse IDE (BG1.2). These results are consistent with the contact with these tools and environments throughout their academic path. Additionally, the subjects have also expressed possessing very little experience in using the Trac software platform (BG1.4), using Trac's Adaptive Software Artifacts Plugin (BG1.5) and using the JHotDraw framework (BG1.7). This was also expected as, unlike the tools and environments that we have mentioned before, the subjects were not lead to use any of these tools during their course.

8.4.2 Platform Activity

The Adaptive Software Artifacts Plugin logged detailed usage information, allowing to determine how much time was spent on wiki pages, on using the *search* feature, on viewing adaptive artifacts, or on accessing the adaptive artifacts' index.

		Duration		
	Module	Σ	$\bar{x}$	σ
Control Group	Wiki Pages	29:09:32	1:32:04	0:51:35
	Search	1:43:17	0:05:26	0:08:45
	ASA Artifacts	0:00:00	0:00:00	0:00:00
	ASA Index	0:00:00	0:00:00	0:00:00
Experimental Group	Wiki Pages	12:05:07	0:48:20	0:27:40
	Search	0:04:45	0:00:19	0:00:21
	ASA Artifacts	4:12:01	0:16:48	0:08:08
	ASA Index	1:11:11	0:04:44	0:05:18

Table 8.4: Descriptive statistics of the time spent on the Trac environment, by module and treatment group. The full dataset can be found on the Web address `http://bit.ly/1iP5Oaj`.

A summary of the time spent on the platform in each of these different platform modules is provided in Table 8.4 and Figure 8.2. It's clear the predominance of *Wiki Pages* as a means of obtaining information and the use of the other areas of the platform is comparatively low. It's interesting to note the differences between the two groups, with the mean of the time spent consuming *Wiki Pages* being almost half in the Experimental Group. Also of note is the fact that the *Search* feature has seen less use by the Experimental Group. The absence of use of *ASA Artifacts* and the *ASA Index* page by the Control Group is quite expected, as these features were only made available to the Experimental Group.

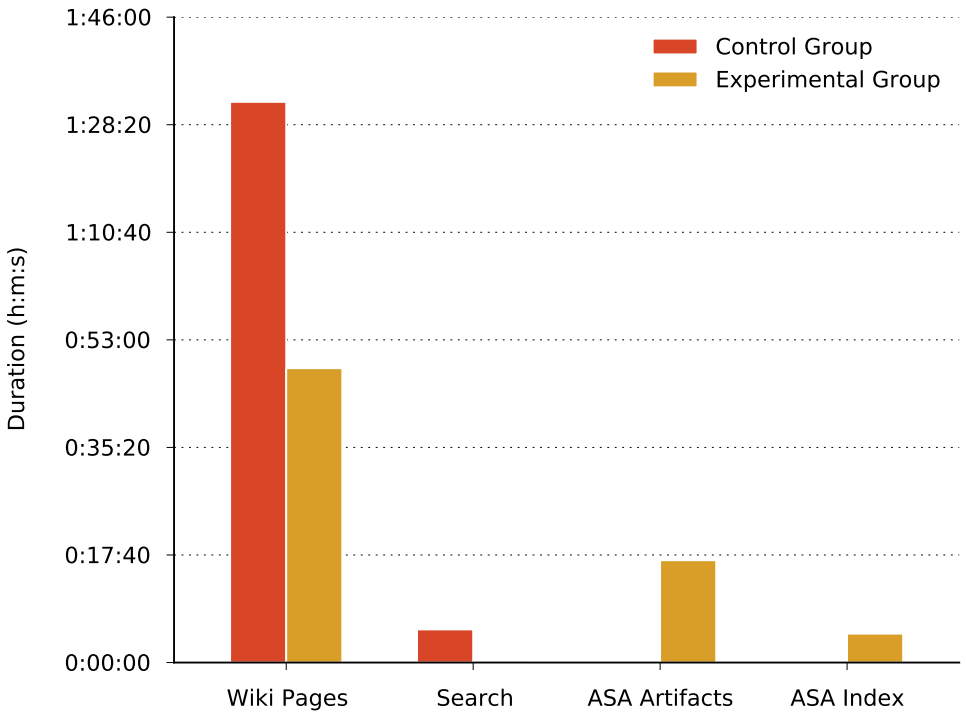


Figure 8.2: Mean of the times spent on the platform by each subject, by platform module. A boxplot chart of the same information can be found on Appendix E and provides finer details.

One-tailed independent samples *t*-tests were used to verify our intuitions and the results are presented in Table 8.5. Note that only the use of Wiki Pages and the Search feature were considered, as the other activities were only available to the Experimental Group.

MODULE	H_1	T	ρ
WIKI PAGES	>	3.073	0.002
SEARCH	>	2.475	0.012

Table 8.5: Result of one-tailed (H_1: CG > EG) independent samples' *t*-tests for the equality of means of the times spent on two platform modules.

The ρ-value obtained through the *t*-test is lower than 0.05 for each of the activities, allowing us to conclude that the values of the Control Group are **significantly greater** than those of the Experimental Group. We can thus conclude that the CG spent more time using these features in the platform, but we may still wonder if the overall time spent in the platform was also greater in the CG.

That can be confirmed by taking the total durations into account. Table 8.6 shows the total durations spent on the platforms and suggests that the Experimental Group has spent comparatively less time consuming information, more specifically, in average, less 27 minutes and 19 seconds. To validate this conclusion, we have also computed a one-tailed independent samples *t*-test that compared the overall durations. These results are presented in Table 8.7 – the obtained ρ-value is lower than 0.05, allowing us to conclude that the time spent in the platform by the Control Group is indeed **significantly greater** than that spent by the Experimental Group.

	Total Duration		
	Σ	$\bar{x}$	σ
Control Group	30:52:51	1:37:31	0:55:07
Experimental Group	17:33:09	1:10:12	0:25:12

Table 8.6: Descriptive statistics of the total times spent on the Trac environments by treatment group. The full dataset can be found on the Web address `http://bit.ly/1iP5Oaj`.

H_1	t	ρ
$>$	1.866	0.037

Table 8.7: Result of one-tailed (H_1: CG > EG) independent samples' *t*-test for the equality of means of the times spent on the Trac environments.

These results point towards the existence of benefits in knowledge acquisition when using the documentation built with the plugin (I1) but they don't allow conclusions to be drawn on the more specific issues stated in Section 8.1 (I1.1, I1.2, I1.3). However, using the assumption that Wiki Pages and ASA Artifacts played a role mostly on *understanding* the contents, and that the use of the Search and the ASA Index mostly supported *finding* the contents, we are able to provide additional insight into issues I1.1[4] and I1.3[5].

Table 8.8 shows the time spent on the platform aggregated by these two activities. There is little difference in the mean time spent finding the contents, but we can see a difference of nearly 27 minutes in the means of the time spent understanding the

[4] **I1.1. Efficiency of acquiring information structure.**
Do developers spend less *time* understanding the contents?

[5] **I1.3. Quality of the classification scheme.**
Do developers spend less *time* searching for the contents they need?

contents. One-tailed independent samples *t*-tests were used to verify the significance of these differences. The results are presented in Table 8.9 and show that the mean time spent *understanding* the contents (I1.1) is indeed **significantly greater** for the CG when compared with the EG, and that there is **no significant difference** in the mean times spent *finding* the contents (I1.3).

		Duration		
	Activity	Σ	$\bar{x}$	σ
Control Group	Understanding	29:09:33	1:32:04	0:51:35
	Finding	1:43:17	0:05:26	0:08:45
Experimental Group	Understanding	16:17:11	1:05:08	0:26:30
	Finding	1:15:57	0:05:03	0:05:16

Table 8.8: Descriptive statistics of the time spent on the Trac environments, by activity and treatment group. The full dataset can be found on the Web address `http://bit.ly/1iP5Oaj`.

These results allow additional conclusions to be reached. Even though the subjects of the EG did spend less time acquiring contents than the CG (Table 8.7), the result of the *t*-test presented in Table 8.9 shows the difference can be attributed to time spent trying to understand the contents (I1.1) and not to time spent trying to find them (I.3). It is interesting to note, though, that the EG spent significantly less time using the Search feature than the CG (Table 8.5), and thus preferred to find the contents they needed by browsing the index of adaptive artifacts.

Activity	H_1	T	ρ
Understanding	>	1.914	0.033
Finding	>	0.149	0.441

Table 8.9: Result of the one-tailed (H_1: CG > EG) independent samples' *t*-tests for the equality of means of the times spent in the platform by type of activity.

8.4.3 Task Durations

Logging the platform activity allows to have a perception of the relative times spent finding, reading and understanding the contents, but it doesn't necessarily reflect the global impact on productivity and task execution. With that in mind we have also looked specifically at the implementations and the times taken by the subjects to achieve them.

The source code was analyzed to make sure that the tasks reported by the participants as finished were in fact fully implemented. We have found that the source code is mostly in agreement to what was reported by each subject. Even though a few details were found missing on some implementations, we consider they had a negligible impact on the development time. Appendix C contains the tasks' data, including the information of which of them were completed – fully, partially, or not at all – and the times taken to conclude each of them, as recorded by the subjects. The duration of each task is summarized in Table 8.10 and in Figure 8.3, where one may see consistently smaller mean durations for all the tasks implemented by the Experimental Group.

		Duration		
	Task	Σ	$\bar{x}$	σ
Control Group	T1	6:56:00	0:21:53	0:12:37
	T2	2:17:00	0:08:03	0:03:05
	T3	3:33:00	0:12:31	0:07:56
	T4	5:46:00	0:20:21	0:08:48
	T5	0:44:00	0:05:30	0:01:43
Experimental Group	T1	5:22:00	0:21:28	0:13:12
	T2	1:43:00	0:07:55	0:05:25
	T3	2:24:00	0:12:00	0:05:24
	T4	2:52:00	0:15:38	0:07:16
	T5	0:33:00	0:03:40	0:02:09

Table 8.10: Descriptive statistics of the time spent on each task. The full dataset can be found on Appendix C.

One-tailed independent samples t-tests were used to confirm the hypothesis that the times spent by the Control Group executing the tasks were significantly greater than those spent by the Experimental Group (H_1: CG > EG). The results are presented in Table 8.11 and they show that, except for task T5, the durations of the Control Group were **not significantly greater** than those of the Experimental Group, despite what a more superficial analysis could lead us to believe.

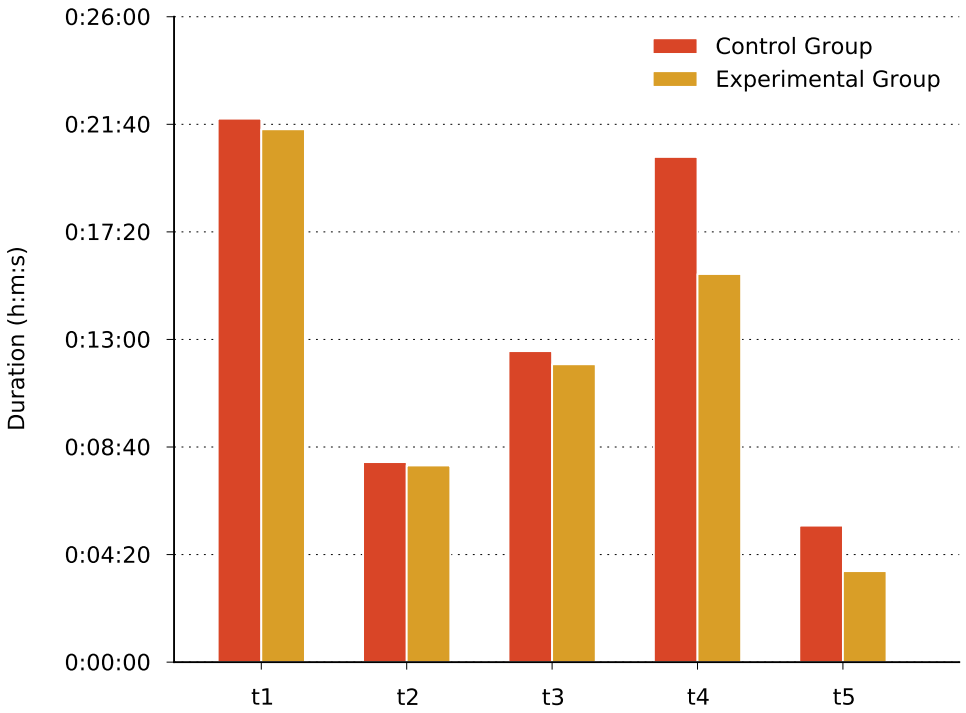


Figure 8.3: Mean of the times spent on each task by each subject, by task. A boxplot chart of the same information can be found on Appendix E and provides finer detail.

Task	H_1	t	ρ
T1	>	0.093	0.463
T2	>	0.078	0.469
T3	>	0.206	0.419
T4	>	1.481	0.076
T5	>	1.823	0.044

Table 8.11: Result of the one-tailed (H_1: CG > EG) independent samples' t-tests for the equality of means of the times spent on each task by each subject.

A similar analysis may be done for the total time spent by the subjects implementing the five tasks. Table 8.12 shows the total durations spent implementing the tasks and suggests that the EG spent comparatively less time. We have also used a one-tailed independent samples t-test in this case to compare the overall durations and the results are presented in Table 8.13. The obtained ρ-value is lower than 0.05, allowing us to conclude that the time spent by the CG implementing the totality of the tasks is indeed **significantly greater** than that spent by the EG.

	Total Duration		
	Σ	$\bar{x}$	σ
Control Group	19:16:00	1:00:50	0:14:53
Experimental Group	12:54:00	0:51:36	0:12:09

Table 8.12: Descriptive statistics of the total time spent by each subject completing the tasks. The full dataset can be found on Appendix C.

H_1	t	ρ
$>$	1.932	0.031

Table 8.13: Result of one-tailed (H_1: CG > EG) independent samples' t-test for the equality of means of the total times spent completing the tasks.

These results don't allow conclusions to be drawn on the specific issues highlighted in Section 8.1 (I1.1, I1.2 and I1.3) but they show that the time spent by the CG implementing the tasks is greater than that spent by the EG (I1). This is in line with the analysis of the platform activity and should be expected if we consider that the time spent on the tasks will include a significant part of the time spent on the platform – the time spent on the platform also includes a setup time, used by the subjects to understand the information that was being made available to them before the tasks started, as well as some additional minutes in the end of the experiment to upload the developed source code.

8.4.4 Assessment Questionnaire

Like the results of the background questionnaire, the results of the assessment questionnaire were analyzed using two-sample Mann–Whitney U statistical tests [HW99]. Tables 8.14 to 8.19 summarize the answers and the results of the tests. The full dataset

can be found in Appendix C and supporting calculations and analysis are included in Appendix E.

Two-tailed or one-tailed Mann–Whitney U tests were used depending on the alternative hypothesis underlying each question. Namely, H_1: CG $\neq$ EG when testing for differences between the control group and the experimental group, H_1: CG < EG when testing for a control group lower than the experimental group, and H_1: CG > EG when testing for a control group greater than the experimental group.

External Factors

This category of questionnaire items was used primarily to discard some threats to validity. The two treatment groups should be exposed to the same context during the experiment and it's important to discard any undesired differences in the physical and digital environments that may have an influence on the results (EF1–EF3, H_1: CG $\neq$ EG).

	CG			EG			MW-U		
	1 2 3 4 5	$\bar{x}$	σ	1 2 3 4 5	$\bar{x}$	σ	H_1	u	ρ
EF1		1.526	0.595		1.267	0.573	$\neq$	178.500	0.142
EF2		1.421	0.815		1.867	1.024	$\neq$	109.500	0.185
EF3		1.421	0.815		1.533	0.806	$\neq$	131.500	0.650

EF1. The room environment was distracting.
EF2. I found difficulties using the IDE.
EF3. I found difficulties using the Java language.

Table 8.14: Summary of the answers to the EF items of the assessment questionnaire, including histograms and descriptive statistics for each question and the result of comparing the answers of the Control Group (EG) using Mann–Whitney U (MW-U) statistical tests.

Running the experiment simultaneously with all the subjects of a group helped making sure that all were under to the same conditions, but we were concerned that some subjects could be distracted by the presence of other people in the room. Another concern was that some of the participants would have trouble or even feel blocked with a problem related to the use of the Java Language or the Eclipse IDE.

However, the answers suggest that the subjects did **not** feel disturbed by the room environment nor had any note worthy issue with the development tools. Let H_1: CG $\neq$ EG, the ρ-values for the three tests (EF1–EF3) reveal that there is **no significant difference** in the answers of the two groups, to any of the three questions.

Overall Perception

The items of this category provide a high-level view of how subjects felt about the platform and the tasks. Other questions may help us conclude about more specific matters and we expect these questions may help us put them into a wider context. The alternative hypotheses are that the experimental group could maybe consider their participation as generally more rewarding than the control group (OP1–OP4, H_1: CG < EG).

	CG			EG			MW-U		
	1 2 3 4 5	$\bar{x}$	σ	1 2 3 4 5	$\bar{x}$	σ	H_1	u	ρ
OP1		3.211	0.893		3.400	1.254	<	121.500	0.228
OP2		3.316	1.029		3.133	1.360	<	159.500	0.734
OP3		3.789	0.893		3.400	1.200	<	166.500	0.813
OP4		3.842	0.987		3.933	1.289	<	125.000	0.265

OP1. I found it easy to translate my knowledge of the problem domain to a concrete solution.
OP2. The project's documentation was easy to use.
OP3. The tasks descriptions were easy to understand.
OP4. I enjoyed the programming exercise.

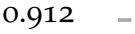
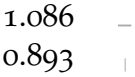
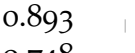
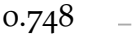
Table 8.15: Summary of the answers to the OP items of the assessment questionnaire, including histograms and descriptive statistics for each question, and the result of comparing the answers of the Control Group (EG) using Mann–Whitney U (MW-U) statistical tests.

However, the ρ-values of the tests for the four questions (OP1–OP4) show that there is **no significant difference** that allows us to conclude that the overall perception of the EG concerning the programming tasks was more positive than that of the CG.

Information Attributes

These questions try to explain how the subjects saw the available information. They focus on knowledge acquisition in general (I1), trying to provide insights on how (the perception of) the information was different with our approach, and don't directly support answering the more specific research issues (I1.1, I1.2 and I1.3). Subjects were asked to deal with a substantial quantity of documentation during the experiment and we postulated that the experimental group would be able to cope better with it and recognize some of its positive attributes. Namely, that they would be able to find all that they needed to know within the information that they were given to complete the tasks (IA1, H_1: CG < EG) and that they would be able to navigate this body of information easily and not be overwhelmed by its dimension (IA2, H_1: CG

< EG). We trusted that both groups would consider the documentation of generally good quality but hypothesized that the combined factors of our approach could make the EG perceive it as better (IA3, H_1: CG < EG). Additionally, there are two information attributes that we were especially interested on – the precision (IA4, H_1: CG < EG) and the concision (IA5, H_1: CG < EG) of the contents.

	CG			EG			MW-U		
	1 2 3 4 5	$\bar{x}$	σ	1 2 3 4 5	$\bar{x}$	σ	H_1	u	ρ
IA1		3.895	0.912		3.733	1.482	<	138.000	0.442
IA2		3.368	1.086		3.467	1.204	<	133.500	0.379
IA3		3.789	0.893		3.800	0.980	<	141.000	0.485
IA4		3.421	0.748		3.467	1.147	<	133.500	0.378
IA5		3.105	0.718		3.733	1.236	<	89.000	0.028

IA1. The information that was made available was in sufficient quantity.
IA2. The information that was made available was not in excessive quantity.
IA3. The information that was made available was of good quality.
IA4. The information that was made available was very precise (i.e., accurate; objective)
IA5. The information that was made available was very concise (i.e, terse; succinct)

Table 8.16: Summary of the answers to the IA items of the assessment questionnaire, including histograms and descriptive statistics for each question, and the result of comparing the answers of the Control Group (EG) using Mann–Whitney U (MW-U) statistical tests.

We found **no significant difference** in the scores of the CG and EG for the first four questions (IA1–IA4), but interestingly found a **significant difference** in the *concision* perceived by the two groups (IA5).

Classification

The questionnaire items in this category helped assessing how the subjects were able to find the contents they needed, thus focusing primarily on issue I1.3[6]. We theorized that our approach would allow subjects to reach the information that they needed more easily (CL1, H_1: CG < EG) and that subjects could perceive that as a result of how the information was organized and linked (CL2, H_1: CG < EG). We expected the EG to resource especially to browsing the available contents (CL3, H_1: CG < EG) and the CG to resource to Trac's search feature (CL4, H_1: CG > EG).

Contrarily to the expected, the result of the statistical tests revealed **no significant difference** in the scores of the CG and EG specifically in what concerns how easy the

[6] **I1.3. Quality of the classification scheme.**
Do developers spend less *time* searching for the contents they need?
See Section 8.1.

	CG			EG			MW-U		
	1 2 3 4 5	$\bar{x}$	σ	1 2 3 4 5	$\bar{x}$	σ	H_1	u	ρ
CL1		3.053	0.999		3.133	1.147	<	138.000	0.441
CL2		3.368	0.809		3.400	1.200	<	135.500	0.407
CL3		3.000	1.170		3.733	1.181	<	89.500	0.030
CL4		3.211	1.321		2.667	0.943	>	178.500	0.097

CL1. I could easily find the information that I needed.
CL2. The way in which the information was organized and linked allowed me to find it more easily.
CL3. I found what I needed to know by browsing the available contents.
CL4. I found what I needed to know by using Trac's Search feature.

Table 8.17: Summary of the answers to the CL items of the assessment questionnaire, including histograms and descriptive statistics for each question, and the result of comparing the answers of the Control Group (EG) using Mann–Whitney U (MW-U) statistical tests.

subjects perceived information to be found (CL1, CL2). The EG found itself reaching the contents they needed especially by browsing the platform, which showed as a **significant difference** in the scores of the two groups (CL3). We could also see a suggestive difference in the means of the two groups in the use of Trac's search feature (CL4), but could find **no significant difference** between them.

Understandability

These questionnaire items allowed gaining insights on how easy it was for subjects to understand the information provided by the platform. They have the goal of supporting an answer to issue I1.1[7]. Most of all, we expected information to be easier to understand by the EG (UN1, H_1: CG < EG) and that it could be perceived as a consequence of how the information was organized and linked (UN2, H_1: CG < EG)

We found a **significant difference** in the scores of the CG and EG that allows us to conclude that the EG did find it easier to understand the contents (UN1). The subjects did not, however, perceived this as a result of how the contents were organized and linked (UN2), as **no significant difference** was found for this questionnaire item.

[7] **I1.1. Efficiency of acquiring information structure.**
Do developers spend less *time* understanding the contents?
See Section 8.1.

	CG			EG			MW-U		
	1 2 3 4 5	$\bar{x}$	σ	1 2 3 4 5	$\bar{x}$	σ	H_1	u	ρ
UN1		3.263	0.909		3.800	0.748	<	94.500	0.038
UN2		3.737	0.636		3.867	0.884	<	134.500	0.390

UN1. The information that was made available was always easy to understand.
UN2. The way in which the information was organized and linked allowed me to understand it more easily.

Table 8.18: Summary of the answers to the UN items of the assessment questionnaire, including histograms and descriptive statistics for each question, and the result of comparing the answers of the Control Group (EG) using Mann–Whitney U (MW-U) statistical tests.

Consistency

The *consistency* questionnaire items supported understanding the subjects' perception regarding the consistency of the contents. Even though the importance of this issue appears especially in the context of the creation of contents and the maintenance of their consistency, we wondered if developers would also perceive consistency differently during the *use* of the contents. In particular, we wondered if the EG would perceive the contents as more consistent (CO1, H_1: CG > EG) and if the two groups would equally consider themselves as having a good perception of the consistency of the contents (CO2, H_1: CG $\neq$ EG). These questions thus had the goal of answering the research issue I1.2[8].

	CG			EG			MW-U		
	1 2 3 4 5	$\bar{x}$	σ	1 2 3 4 5	$\bar{x}$	σ	H_1	u	ρ
CO1		2.368	0.985		1.800	0.748	>	187.500	0.052
CO2		2.368	1.086		2.400	1.143	$\neq$	138.000	0.886

CO1. The information that was made available was often inconsistent.
CO2. I don't have a good perception if the information that was available to me was consistent or not.

Table 8.19: Summary of the answers to the CO items of the assessment questionnaire, including histograms and descriptive statistics for each question, and the result of comparing the answers of the Control Group (EG) using Mann–Whitney U (MW-U) statistical tests.

The means of how the two groups perceived consistency suggested that the EG perceived the contents as more consistent than the CG, but the test found **no significant difference** between the two groups (CO1, CO2).

[8] **I1.2. Consistency of the contents.**
Are resulting contents more *consistent*?
See Section 8.1.

8.5 Validation Threats

The goal of this experiment is to gather evidence supporting scientifically sound answers to the research issues. But some experimental conditions may cast doubt over the validity of that evidence and call for a closer look. These conditions are not part of the *treatments* – i.e., it's not interesting or useful to see them as part of the *causes* in the cause-effect relationships that we are trying to identify – but they have the potential to cause deviations in the results. Some steps were taken during the design of the experiment that allow us to discard some of these threats to validity. Others need to be taken into account during data analysis in order not to over-generalize the results.

Placebo Effect. The placebo effect consists of a positive perception towards a particular treatment, caused solely by the subjects' belief on the treatment's benefits. We were concerned that differences in the results of the questionnaires could not be exclusively a product of the different treatments, and that this effect could induce unwanted deviations.

The participants had already learned during the recruitment phase that they would be participating of an experiment and during the *introduction* step of the procedure they were instructed about the environments that they were about to use. We believe this threat can be discarded because the participants were **not** told if and in which way their group would differ from the other group – as far as the participants on the EG knew, they were using exactly the same environment as the CG, and vice-versa.

Different Skills. The design of the experiment assumes that the subjects have roughly the same skills and proficiency with a few tools, like the Java programming language, the Eclipse IDE, software forges, object orientation, software frameworks, software documentation, wikis and the development of GUI applications. The subject's grades and the background questionnaire analyzed in Section 8.4.1 support discarding this threat, as no significant difference could be found between the subjects of the two groups.

External Factors. We were also concerned that something in the physical or digital environment could cause a deviation in the results of the two groups. The *external factors* items of the assessment questionnaire analyzed in Section 8.4.4 helped to discard this threat to validity, as there was no significant difference between the two groups and no relevant operational problems of such sort were found.

Lack of Engagement. A lack of motivation by the subjects to perform well in the experiment could bias the results. Foreseeing this concern, we have provided an incentive to participate of the experiment in the form of a small bonus on one of the subjects' courses. We thus believe that we can discard this as a validation threat.

Insufficient Sample Size. Although there are a few rules of thumb when determining the sample size, it's in practice most often a product of the available resources and the researchers' ability to recruit subjects for the experiment. We would certainly like to see this user-study replicated using a larger sample size, as it would increase its power and help to discard this threat to validity more decisively.

Non-Generalizability to Professionals. Having students as subjects had the benefit of providing a more homogeneous set of skills, which made it easier to ensure there were no significant differences between the CG and the EG. The main motivation for recruiting students though was that students were easier to recruit. Taking into account that students are not necessarily fully representative of professional software developers [CJMS10], we have called into question if the current results could be generalizable to professionals. We believe they could but further studies are required to put that hypothesis to the test. The preliminary evidence provided by this study will be important during recruitment, to encourage professionals to participate.

Non-Generalizability to Industrial Projects. The controlled environment required to conduct an experiment necessarily implies a departure from *real-world* conditions, with the goal of studying a specific *cause-effect* relationship. On the other hand, experiments can hardly answer detailed *how* and *why* questions. They are a reductionist approach to understanding a phenomenon. That is, they consider that the object of study can be explained by reducing it to more fundamental parts and the way that they interact. For this reason, experiments are more difficult to conduct in settings with a high amount of variables that interact in many ways.

One particular point of concern relates to how the creation and use of software documentation is often intertwined with software development activities and both will often happen in short feedback loops. By analyzing the *use* and the

creation of software documentation as separate phenomena we are not able to observe the relation between the two activities.

Another example lies in how the starting point for the experiment, namely, the software documentation, had to be adapted (i.e., changed, to some degree) by the researchers to Trac's wiki pages. The software documentation could be more credible if created entirely by professionals in the context of an industrial project.

One must wonder to which extent may the effects on *toy* projects be representative of full-strength industrial applications. To discard this threat, we must combine the results of this experiment with those of other research methods. In particular, we feel that one or more industrial case-studies could help considerably to reduce this threat to validation.

8.6 Summary

This chapter described an experiment conducted in an academic setting to support the validation of the approach described in Chapter 6 through the use of a plugin for the Trac software forge which was described in Chapter 7. The experiment specifically tries to provide insights on those issues related to knowledge acquisition – I1[9], I1.1[10], I1.2[11] and I1.3[12].

The design of the experiment ensures the Control Group and Experimental Group to have subjects of equivalent skill level. It combines the use of data obtained by measuring how subjects have used the platform, with the use of data obtained through questionnaires.

Analysis of platform activity and task durations revealed that the EG did spend significantly less time implementing the several tasks and, particularly, on the platform searching and consuming its contents. This suggests benefits in knowledge acquisition when using the documentation built with the plugin and we were able to attest a significant difference in the times spent in the platform understanding the contents between the CG and the EG (I1 and I1.1).

[9] **I1. Efficiency of Knowledge Acquisition.**
Do developers spend less *time* acquiring knowledge from the contents?

[10] **I1.1. Efficiency of acquiring information structure.**
Do developers spend less *time* understanding the contents?

[11] **I1.2. Consistency of the contents.**
Are resulting contents more *consistent*?

[12] **I1.3. Quality of the classification scheme.**
Do developers spend less *time* searching for the contents they need?

The assessment questionnaire aimed to complementing these results and provides more specific, albeit subjective, evidence of the benefits of using the Adaptive Software Artifacts approach. It's important to note that these results don't imply a direct answer to the research issues, as no objective measurement of the time spent acquiring contents can be obtained from the answers to the questionnaire – they address each issue by providing additional insights and supporting explanations.

The significant difference found in the answers of the questionnaire item IA5[13] suggests that knowledge was easier to acquire on the EG (I1). This result is also supported by the answers to UN1[14] and CL3[15] which, respectively, provide insights into issues I1.1 and I1.3. In other words, we assume that being able to understand information more easily will reduce the duration of that process (I1.1) and that, having preferred to browse for contents, subjects were expecting it to be a quicker way to find contents when compared with using the platform's search feature (I1.3).

I1.2 was taken into account in the assessment questionnaire in the form of items CO1[16] and CO2[17], but no significant difference was found in the answers to these questionnaire items that supports a conclusion. We were curious about the perception of consistency of the contents by the users, but the contents provided to the two groups were equivalent in everything except their form, which may be enough to explain why no differences in consistency were perceived.

Table 8.20 summarizes the research issues addressed by the experiment and for which of them we have found statistically significant results that support the thesis validation.

	I1	**I1.1**	**I1.2**	**I1.3**
Platform Activity	✓	✓		
Task Durations	✓			
Assessment Questionnaire	✓(IA5)	✓(UN1)		✓(CL3)

Table 8.20: Research issues addressed by the experiment. The lighter check mark shows a general issue that was addressed indirectly, by addressing one of its more specific issues. Issues addressed by the assessment questionnaire are annotated with the specific questionnaire item in question.

The answers provided by these results are promising and encourage us to address those questions still left answered with further user-studies. Validation efforts should

[13] IA5. The information that was made available was very concise (i.e, terse; succinct).
[14] UN1. The information that was made available was always easy to understand.
[15] CL3. I found what I needed to know by browsing the available contents.
[16] CO1. The information that was made available was often inconsistent.
[17] CO2. I don't have a good perception if the information that was available to me was consistent or not.

now focus on the replication of the experiment by more researchers and on discarding outstanding threats to validity. Increasing the sample size and getting the participation of professional software developers on future runs of the experiment can strengthen the confidence on the results and their generalizability.

Chapter 9

Conclusions

This dissertation has put software documentation into the context of knowledge sharing and preservation and related it to how knowledge evolves within software development. It goes on to describe software artifacts – notably, software documentation artifacts – and to describe software evolution – in particular, concerning how software artifacts change and the techniques to make software and its information easier to evolve.

This overview first led us to establish three key concerns that would be important to address: a) the expression of information structure, b) the maintenance of contents' consistency and c) the classification of those contents so that they can be found more easily. It then led us to design the approach that we named as Adaptive Software Artifacts, which combines the benefits of free-form and structured contents and lowers the barrier to adding (and changing) arbitrary structure to textual information.

9.1 Contributions

The definition and approach to the above concerns stemmed from the thesis that:

> *Capturing software knowledge with the Adaptive Software Artifacts approach makes information easier to be consumed, created and evolved, especially in the context of medium-to-large projects.*

This thesis was decomposed into eight research issues, two general issues and six specific ones. They are revisited below. Pursuing solutions to these issues produced four main contributions:

1. A patterns catalog. The patterns were presented in Chapter 5 and document solutions used throughout this research, from the design of the approach itself

(Chapter 6) to the design of the reference architecture and implementation (Chapter 7). They describe such solutions at different abstraction levels and cover different topics, such as software documentation, information classification, flexible modeling tools and adaptive object-models. In total, twenty-five new patterns were described.

2. An approach to software documentation. The Adaptive Software Artifacts approach was described in Chapter 6 and combines benefits of free-form contents and benefits of structured contents. Its main goal is to support greater flexibility by allowing developers to add structure to free-form contents and by leveraging that structure for consistency maintenance and for the classification of textual information. The description of the approach defines a set of design principles and activities, which served as the basis for creating the reference architecture and implementation.

3. A reference architecture and implementation. An implementation of the Adaptive Software Artifacts approach was described in Chapter 7. It assumes the form of a plugin for the Trac software forge. Its creation was motivated by multiple goals; namely, proving the practicability of the approach and serving as a reference for other developers trying to implement it. Additionally, the plugin was developed in view of the empirical validation of the approach and was used in the user-study described in Chapter 8.

4. A statistical experiment designed to validate the approach. A user-study, in the form of a statistical experiment, was conceived to validate empirically the effect of the Adaptive Software Artifacts approach on some of the issues that it aims to address. The experiment was described in detail in Chapter 8. It focuses specifically on knowledge acquisition issues and was designed to account for two experimental groups – a control group, using a regular Trac software forge, and an experimental group, using a Trac software forge enabled with the plugin. It ensures that both groups have an equivalent skill level and uses data obtained from the users' activity in the platform, from the duration of the tasks of the experiment, and from questionnaires. This design can, and should, be used to replicate the experiment independently. This run of the experiment in an academic setting produced some results that reveal benefits in knowledge acquisition when using documentation that follows the approach.

The eight research issues introduced in Chapter 4 are listed in Table 9.1, which summarizes the extent to which each issue is addressed and the extent to which the approach showed statistically significant benefits. The focus of the experimental design is on knowledge acquisition and some of the benefits were found in the efficiency of acquiring knowledge in general (I1) and, more specifically, in the acquisition of information structure (I1.1) and in the use of the provided classification scheme (I1.3).

Research Issues	Addressed	Significant Benefits
I1. Efficiency of Knowledge Acquisition	✓	✓
I1.1. Efficiency of acquiring information structure	✓	✓
I1.2. Consistency quality of the contents	✓	–
I1.3. Quality of the classification scheme	✓	✓
I2. Efficiency of Knowledge Capture	✗	–
I2.1. Efficiency of expressing information structure	✗	–
I2.2. Economy of consistency maintenance	✗	–
I2.3. Economy of classification scheme maintenance	✗	–

Table 9.1: Research issues and the extent to which they were addressed and validated by the statistical experiment.

An additional contribution and validation of the work presented in this dissertation lies in the papers reviewed and accepted for publication by the research community. The final pages of this thesis (p. 231) include the complete list of published papers.

An ancillary contribution may be found in *Weaki*, the prototype of a wiki engine developed to test the feasibility of some early ideas. Weaki has the goal of supporting the incremental capture and evolution of free-text document artifacts through a notion of *weakly-typed pages* [CFFA09a] that is very similar to the notion of lightly-constrained templates described in Section 2.3.2. The development of Weaki focused on dealing with evolving free-text document structures[1] and increasing awareness by the users towards that kind of structure of the contents. Since its conception, the notion of weakly-typed wiki pages has already been further explored by more authors [Kac12]. Weaki's design is inspired by the design of weakly-typed programming languages and it applies to the layout structure of the contents the founding principles of wikis [Cunb]. Although the concept isn't necessarily tied to software development, it makes good sense to apply it to this domain as a supplement to the Adaptive Software Artifacts

[1] It is important to make the distinction between the structure of a free-text document and domain structure. While the first confers the contents only a layout form, the second describes and adds new information to the contents themselves.

Plugin, where it may be used to improve the suggestion of new adaptive software artifacts from free-text document structure (A8, p. 127).

9.2 Future Work

Doctoral works are always subject to a time horizon, and new goals and directions will easily emerge during the path of any scientific research. These two factors conspire to provide a constant backlog of ideas for future work, as described by the following sections.

Connections to Related Areas

This work positions itself primarily in the domain of software engineering, which is the researcher's main area of training and experience, but it is not difficult to find connections between how knowledge is handled in this domain and many other areas where *knowledge work* is prevalent [Dav05]. Additionally, it is easy to find connections between our work and the cognitive sciences and information science domains, suggesting that it may be possible to find and/or put into practice deeper, overarching, theories of how knowledge capture and acquisition should be supported for efficiency and ease of evolution.

It would be interesting to explore these connections further, possibly by collaborating with researchers in the cognitive sciences and information science domains. We are curious as to whether the conclusions of this work may be applicable to other fields.

Patterns

The patterns presented in Chapter 5 don't cover the entire problem and solution spaces and many more could still be added to this catalog. Completing the catalog, with patterns at different abstraction levels, could make the four categories of patterns that we have addressed connect more seamlessly and converge to a pattern language for software documentation tools in general and, in particular, those supporting the Adaptive Software Artifacts approach. Two patterns could already be added to the Information Classification category (Section 5.5) – Bottom-up Information Structure and Top-down Information Structure – which would pull this category *closer* to the Flexible Modeling Tools category (Section 5.6). Additional patterns have also

already been identified in the Adaptive Object-Models category (Section 5.7) and may be described and published shortly.

Furthermore, patterns and pattern descriptions evolve over time to include new understandings of their nature and of their *forces*, *consequences*, *contexts* and *known uses* [KP10, BHS07]. This is in line with Christopher Alexander's notion of patterns as described in *A Pattern Language* [Ale77]:

> *"each pattern represents our current best guess as to what [...] will work to solve the problem presented. The empirical questions center on the problem – does it occur and is it felt in the way we have described it? – and the solution – does the arrangement we propose in fact resolve the problem? [...] the patterns are [...] hypotheses [...] and are therefore all tentative, all free to evolve under the impact of new experience and observation"*

The patterns described in this thesis could still be updated with new *known uses*, which would strengthen their validity.

Empirical Validation

Chapter 8 presented a statistical experiment that supported some evidence for the benefits of the approach, but further user-studies will strengthen these results and may be used to validate the research issues that remain unaddressed.

Statistical Experiment

The experimental results presented in Chapter 8 proved promising and it will be interesting to conduct more user-studies to address those questions still left answered. Furthermore, this experiment has focused specifically on knowledge acquisition and has not approached knowledge capture issues – I2[2], I2.1[3], I2.2[4] and I2.3[5] – which are equally important and should also be the focus of empirical studies.

Additionally, independent validation by other researchers will help to discard outstanding threats to validity and conducting the experiment with a bigger sample size composed by professional software developers can strengthen the confidence on the generalizability of the results.

[2] **I2. Efficiency of Knowledge Capture.**
Do developers spend less *time* capturing contents?

[3] **I2.1. Efficiency of expressing information structure.**
Do developers spend less *time* capturing the contents?

[4] **I2.2. Economy of consistency maintenance.**
Do developers spend less *time* doing consistency maintenance?

[5] **I2.3. Economy of classification scheme maintenance.**
Do developers spend less *time* maintaining a classification scheme?

Case Studies

Experiments are very effective to assess cause and effect relationships but they imply controlling everything else in the experiment's environment, which means that factors that may turn out to be relevant in the *real* world may be easily ignored. To deal with this limitation, researchers often combine experiments with other research methods that enable them to improve their knowledge concerning the questions at hand. This motivates us to put the Adaptive Software Artifacts approach to test through industrial case-studies, which we hope will provide additional insights into the benefits and liabilities of this approach. Two software companies have already expressed interest in this work and its results and we believe they may be willing to try the Trac plugin in their contexts.

Data Analysis Framework

The data analysis module described in Sections 8.2.6 and 8.2.8 was developed specifically with the analysis of the experiment in mind. It is a byproduct of this work and has its own usefulness in this context but we feel that it could also become useful beyond this scope. By making it more abstract and evolving it with the goal of using it in other experiments, it's possible to extract from this module an experimental data analysis framework.

Improvements to the Plugin

The development of the plugin described in Chapter 7 focused on the core activities of the Adaptive Software Artifacts approach and on those important for conducting the experiment described in Chapter 8. As such, this implementation doesn't yet encompass all the activities described in Sections 6.3 and 7.2. It would be interesting to make the implementation cover the entire approach to allow other kinds of empirical validation. Additional improvements were introduced in Section 7.7 and should be the next steps in the development.

Appendices

Appendix A

Language Piggybacking Pattern

The Type Square pattern reinvents notions that may already exist in the programming language, like classes, class instances and properties. Its goal is to support a greater flexibility than the language allows, by enabling the end-user to create a confined set of classes and properties herself, at runtime.

This allows overcoming limitations in programming languages' reflection capabilities, as some languages make reflection very cumbersome to use or provide it within a reduced scope. But not all languages exhibit such limitations. Dynamic languages, in particular, usually make reflection available and easy to use.

Problem

When implementing a Type Square*, developers tend to need to reimplement many of the same mechanisms of the programming language that depend on the notions of classes, instances and properties.*

The Type Object pattern is a constituent part of a Type Square. It allows *classes* to be created by the end-user, by defining an `EntityType` class to represent the system's classes and an `Entity` class to represent the system's instances. For these *classes*, the instantiation link – the relation between classes and instances – as provided by the programming language (represented as the `instance of` relation between `type` and `object`) is *replaced* by an association between the `EntityType` class and the `Entity` class – Figure A.1.

When the ability to dynamically create classes is not present or is difficult to use, this approach provides more **flexibility**. On the other hand, it implies a **loss of the features** provided by the programming language that depends on the instantiation link. For example, developers using the python programming language to implement

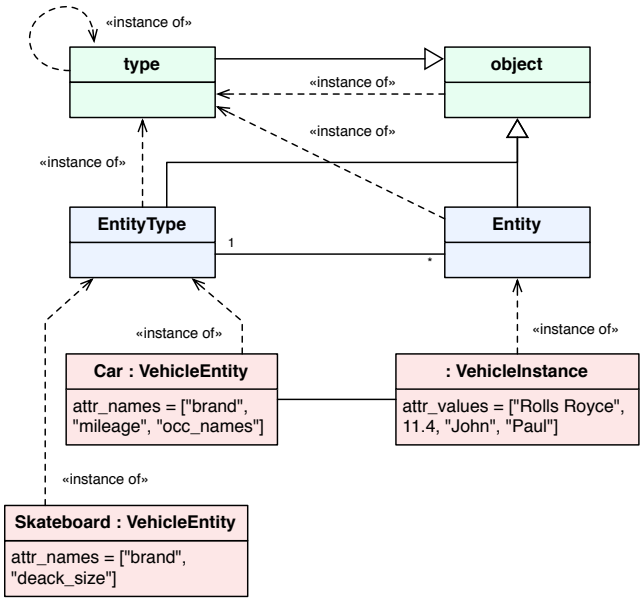


Figure A.1: An implementation of the TYPE OBJECT pattern in the python programming language. `type` and `object` represent classes of python's object model. `EntityType` and `Entity` are roles played by the corresponding classes in the TYPE OBJECT pattern. The diagram uses the notation for UML class diagrams, and different colors to distinguish between different meta levels.

the TYPE OBJECT pattern won't be able to instance a class invoking the class constructor and initializer (e.g., `myobj = MyClass()`) or use the `myobj.__class__` attribute to find what is the class (i.e., the `EntityType`) of a given instance (i.e., of a given `Entity`). An implementation of these and other lost mechanisms can be individually added to the TYPE OBJECT implementation, but imply additional costs in **development effort** and **code complexity**.

Solution

Leverage the language's object model as much as possible, resourcing to its reflection mechanisms if needed, and only extend that model to the extent necessary.

Applying the TYPE OBJECT in conjunction with this pattern could lead to the solution shown in Figure A.2, which can be easily contrasted with Figure A.1.

The instantiation mechanism of the language is used to represent the instantiation link between the instances of `EntityTypes` and the instances of `Entities` – note the *`instance of`* link between `Car` and its instance, mirroring the *`instance of`* link between `object` and `type`. The inheritance mechanism of the language is used to represent the relation between classes and subclasses – note the inheritance link between `Vehicle` and its subclasses, mirroring the inheritance link between `type` and

`object`.

The resulting object graph is similar to what would exist if `Vehicle`, `Car` and `Skateboard` were manually created at design time instead of dynamically created at runtime. The `EntityType` and `Entity` classes are points from which to extend the meta object model – `EntityType` may, for example, provide an `user_readable_name` attribute that would store the name of each `EntityType` in the form in which it could be represented in the user-interface.

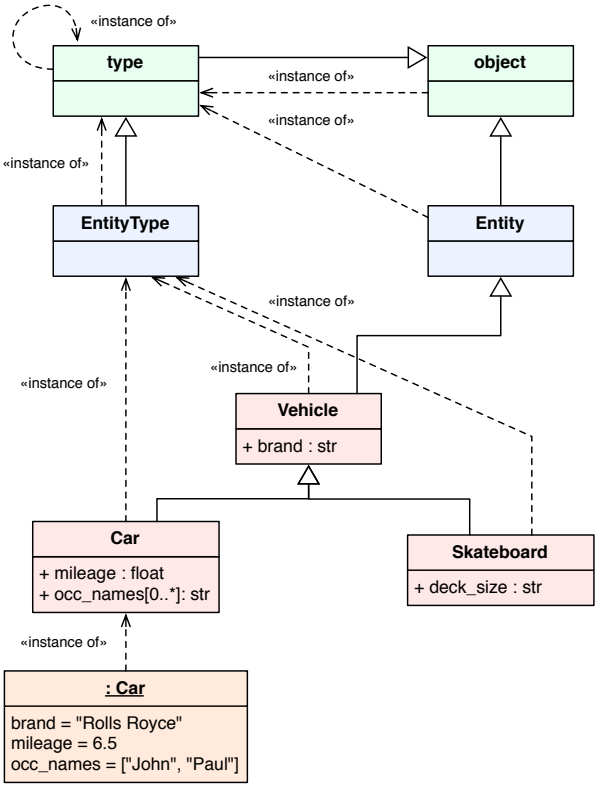


Figure A.2: An example of the Language Piggybacking pattern illustrating an implementation of the Type Object pattern in the python programming language. `type` and `object` represent classes of python's object model. `EntityType` and `Entity` are roles played by the corresponding classes in the Type Object pattern. The diagram uses the notation for UML class diagrams, and different colors to distinguish between different meta levels.

A less positive aspect of this solution is that by binding the meta-model more tightly to that of the programming language, we are limiting the amount of expressiveness at the meta-model level, and thus at all the levels below it. It is more difficult, for example, to forgo the notion of *class* (i.e., `EntityType`) at the meta-model level.

Known Uses

A known use is the Adaptive Software Artifacts Plugin for Trac, which uses the python language's reflection capabilities to allow its users to create and to instance new kinds of object (more specifically, new kinds of *adaptive software artifacts*) at runtime.

Appendix B

Experiment Materials

This appendix includes the materials used in the statistical experiment described in Chapter 8.

Background Questionnaire

Background Questionnaire

Welcome! Thank you for participating in this experiment. Before you start, we will ask you to answer this brief questionnaire about your profile as a developer. It shouldn't take more than a minute.

Use an 'X' to mark the answers that best reflect your opinions according to the scale: **1 (Strongly Disagree), 2 (Somewhat Disagree), 3 (Neither Agree nor Disagree), 4 (Somewhat Agree), 5 (Strongly Agree)**.

BG1. I have considerable experience...

	1	2	3	4	5
...using the Java programming language					
...using the Eclipse IDE.					
...using Software Forges.					
...using the Trac platform.					
...using Trac's *Adaptive Software Artifacts* or *Custom Software Artifacts*.					
...using frameworks.					
...using the JHotDraw framework.					
...with object-oriented software development.					
...extending a system using composition and subclassing.					
...developing industry-level applications.					
...maintaining/modifying industry-level applications.					
...documenting software systems.					
...using technical documentation of software systems.					
...using wikis.					
...developing standalone GUI (Graphical User Interface) applications.					

After filling-in the questionnaire, please store it inside the envelope that you will receive.

Software Engineering Group, College of Engineering, University of Porto

Figure B.1: Background questionnaire, used to assess any statistical deviation between the CG and EG in what concerns their experience or skills.

Participant Instructions

Empirical Study in Software Engineering

In the next 1h20 your job will be to develop a small desktop application. You will be granted access to:

- A workstation with Windows 7, JDK 1.7 and Eclipse 4.2 (Juno)
- A pre-configured eclipse project;
- Some online documentation.

In the end, you should deliver the source-code of the application, ready for use.

Time Tracking

You will be given the description of five (5) tasks. Please use the table below to keep track of the times you start/finish implementing each of them. Note that only the *first* task requires you to record the *start* time; you should record this value only when you start implementing the task, or reading docs with the specific intent to implement it.

	Time Started	Time Finished
T1		
T2	–	
T3	–	
T4	–	
T5	–	

Non-Disclosure Agreement

BY PARTICIPATING IN THIS EXPERIMENT, YOU HEREBY GRANT PERMISSIONS FOR THE USE OF ALL PRODUCED ARTIFACTS, FOR RESEARCH AND DEVELOPMENT ACTIVITIES BY THE SOFTWARE ENGINEERING GROUP — COLLEGE OF ENGINEERING, UNIVERSITY OF PORTO. YOU FURTHER ACKNOWLEDGE AND AGREE THAT ACTIVITIES DURING THIS EXPERIMENT MAY BE DIGITALLY RECORDED AND ARCHIVED.

YOU DECLARE, UNDER YOUR HONOR, THAT YOU'LL NOT PROVIDE ANY KIND OF DETAILS, EITHER DIRECTLY OR INDIRECTLY, ABOUT ANY OF THE CONTENTS OF THIS EXPERIENCE, TO ANYONE ELSE BESIDES THE OTHER ELEMENTS OF YOUR WORKSHOP GROUP AND THE LEAD RESEARCHERS – FILIPE CORREIA, NUNO FLORES, JOÃO PASCOAL FARIA AND ADEMAR AGUIAR – FROM THE SOFTWARE ENGINEERING RESEARCH GROUP. THIS AGREEMENT IS VALID FOR ONE (1) YEAR STARTING FROM THE DATE OF THE EXPERIMENT.

Porto/FEUP, _____ of _____ of 2013

_____
Signature

Ready, Go!

Logon to the project's website, where you will find your instructions. You can find the web address and login details on page 2.

Figure B.2: First page of the instructions sheet given to each participant after the background questionnaire. It was used to a) direct participants to the platform, b) collect the tasks' start and completion times and c) ask the participants to sign the non-disclosure agreement. The second page consisted of the credentials and base URL to log into the platform – it is specific to each subject and hence not included here.

Documentation

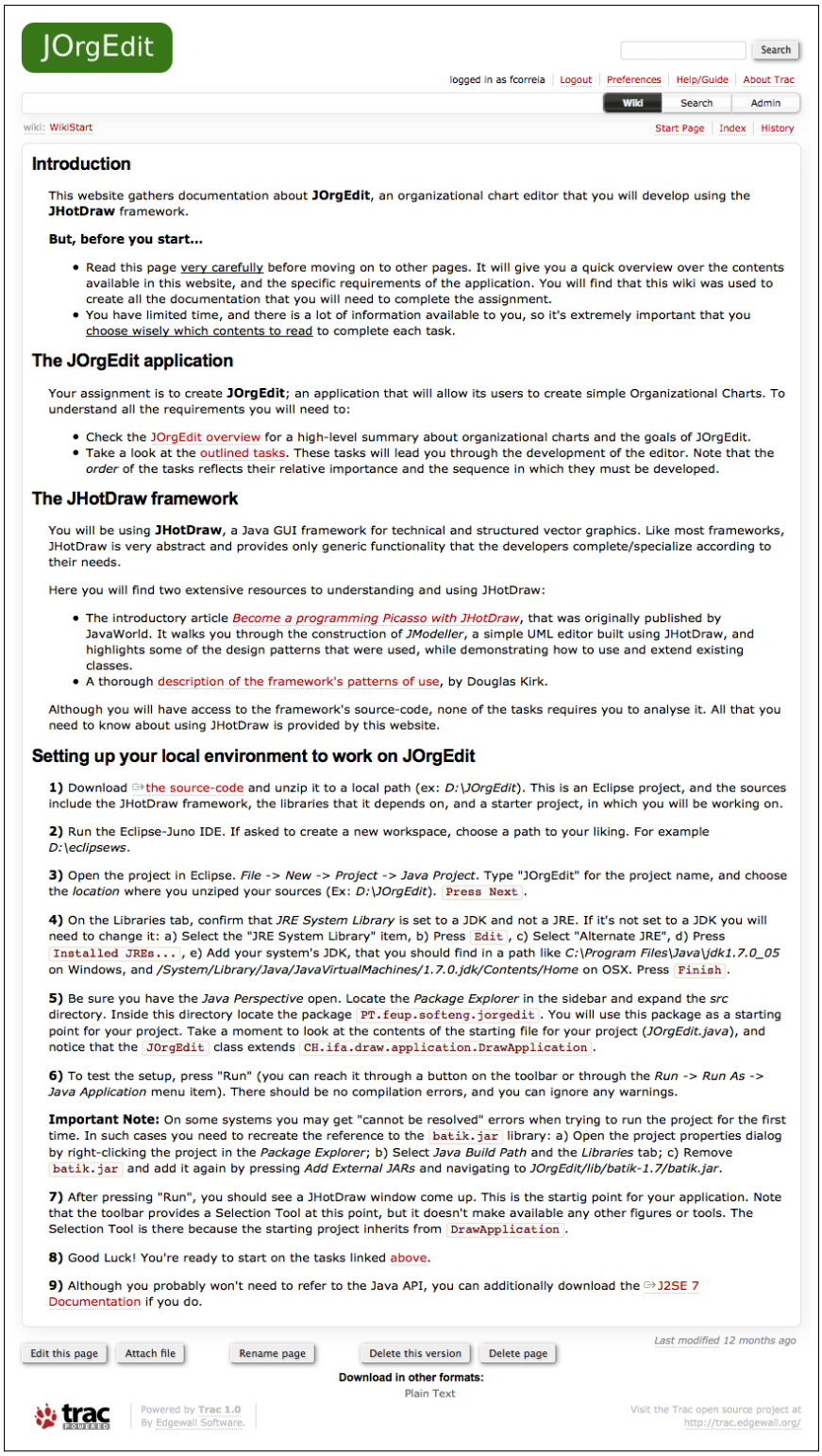


Figure B.3: Opening page of the platform instructions provided to the Control Group. The only difference to the instructions provided to the Experimental Group lies on the absence of the Adaptive Software Artifacts Plugin.

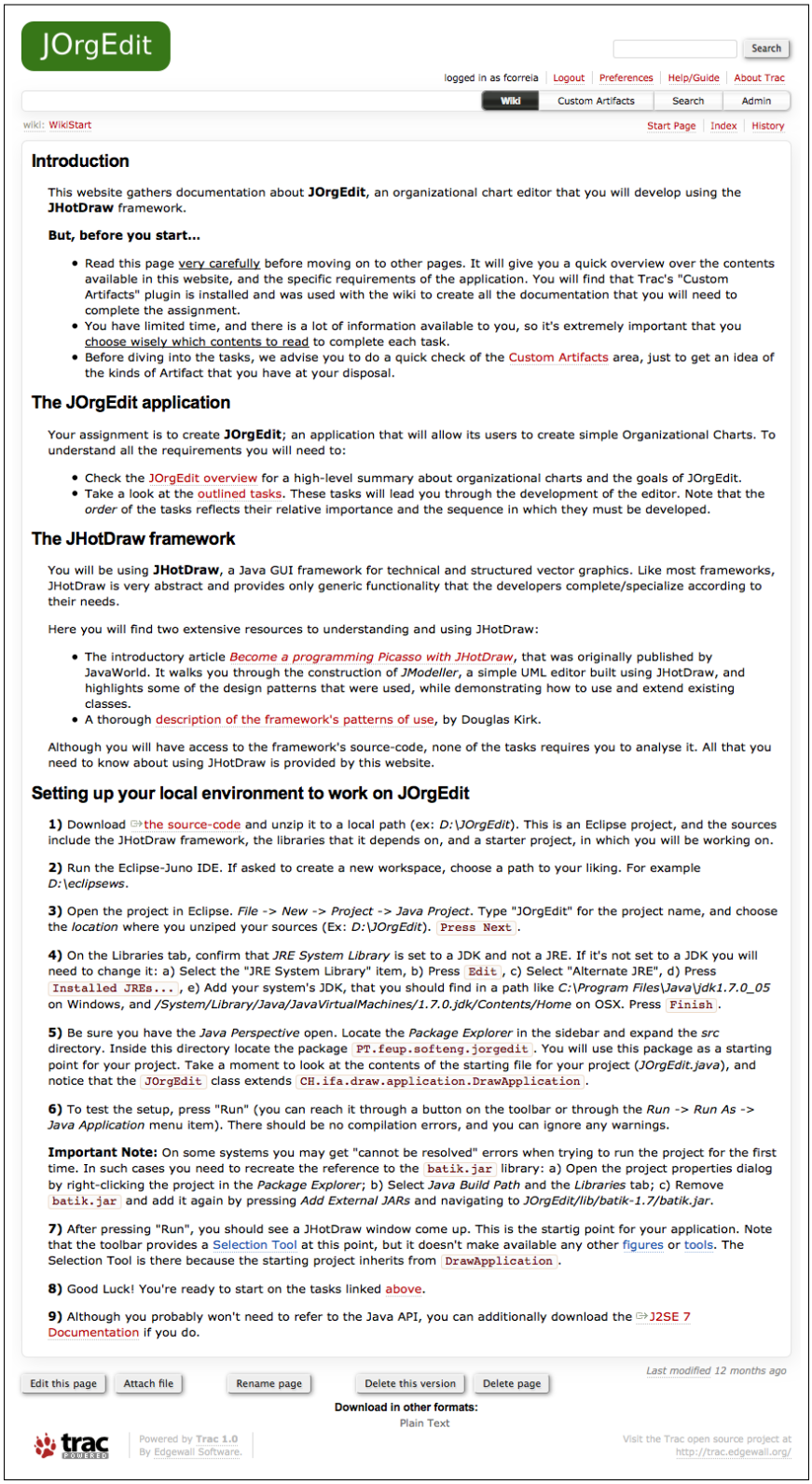


Figure B.4: Opening page of the platform instructions provided to the Experimental Group. The only difference to the instructions provided to the Control Group lies on the presence of the Adaptive Software Artifacts Plugin.

The Trac instances used in the experiment provided all the documentation, including some technical documentation, some domain documentation, and the description of the tasks themselves. Figures B.3 and B.4 show the opening pages of the Trac instances, through which all this documentation was made available, respectively for the control and experimental groups. Figures B.5 to B.7 show part of the task descriptions.

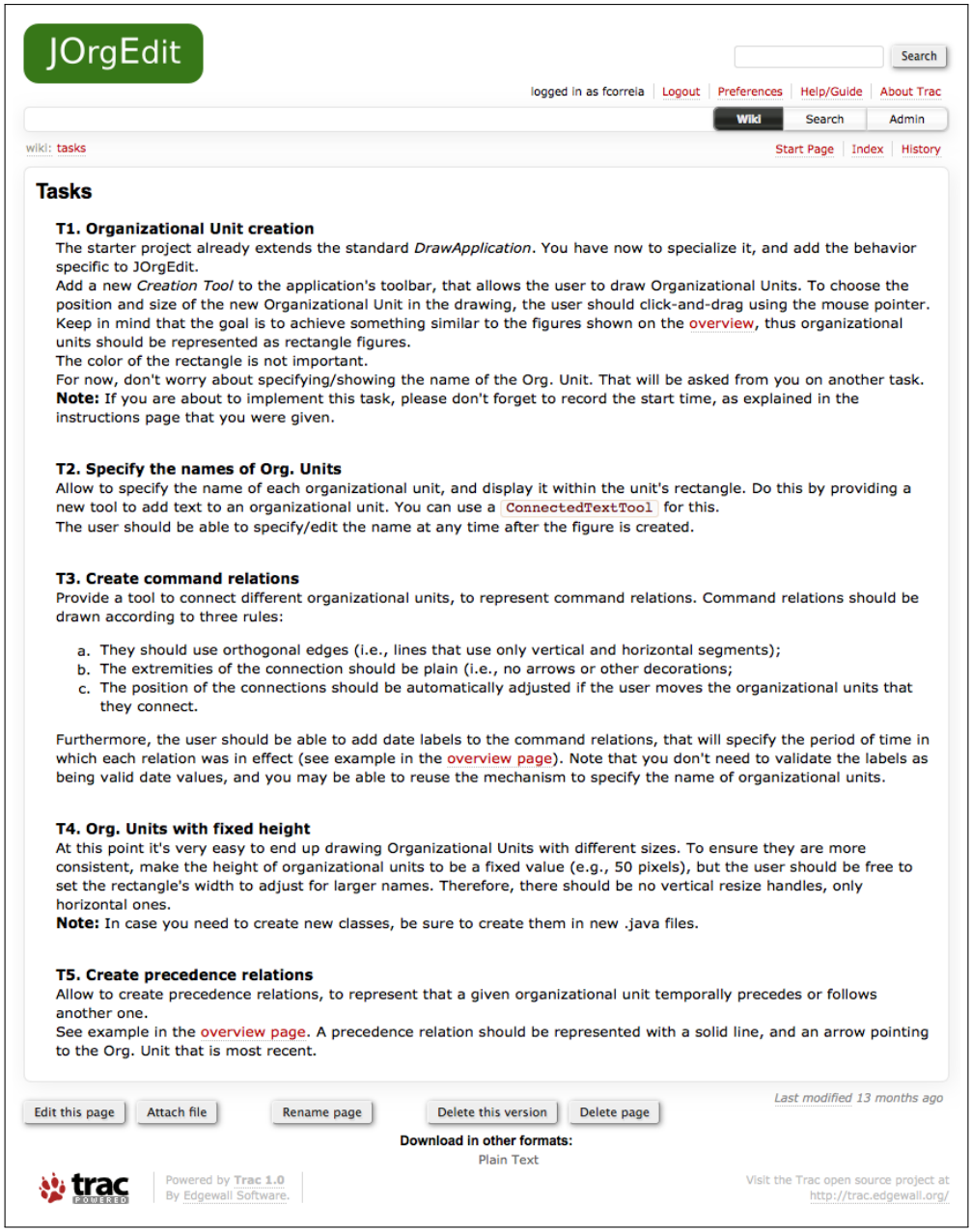


Figure B.5: Tasks page of the platform instructions provided to the Control Group.

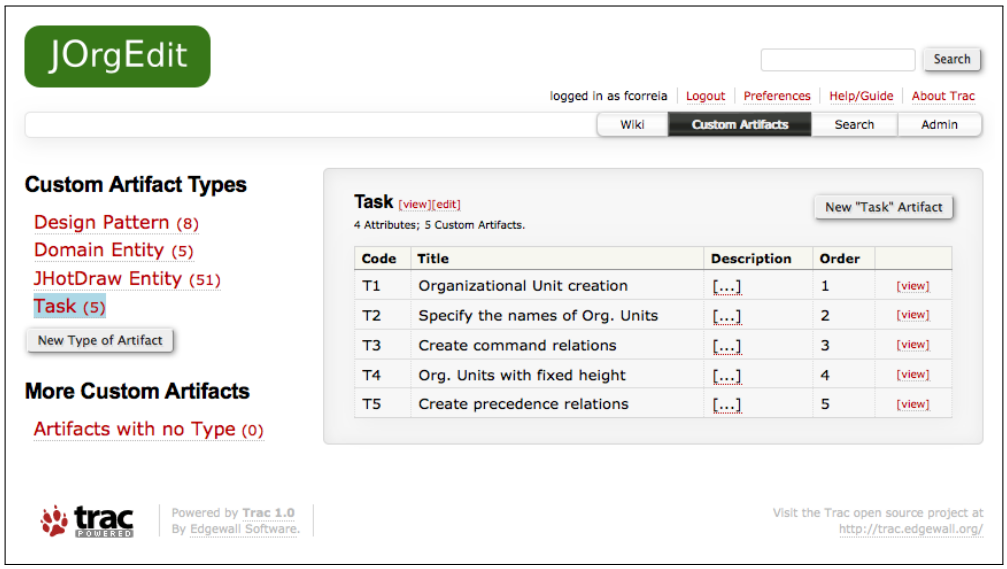


Figure B.6: Tasks page of the platform instructions provided to the Experimental Group. Unlike the task descriptions provided to the Control Group, those provided to the Experimental Group were created using the Adaptive Software Artifacts Plugin.

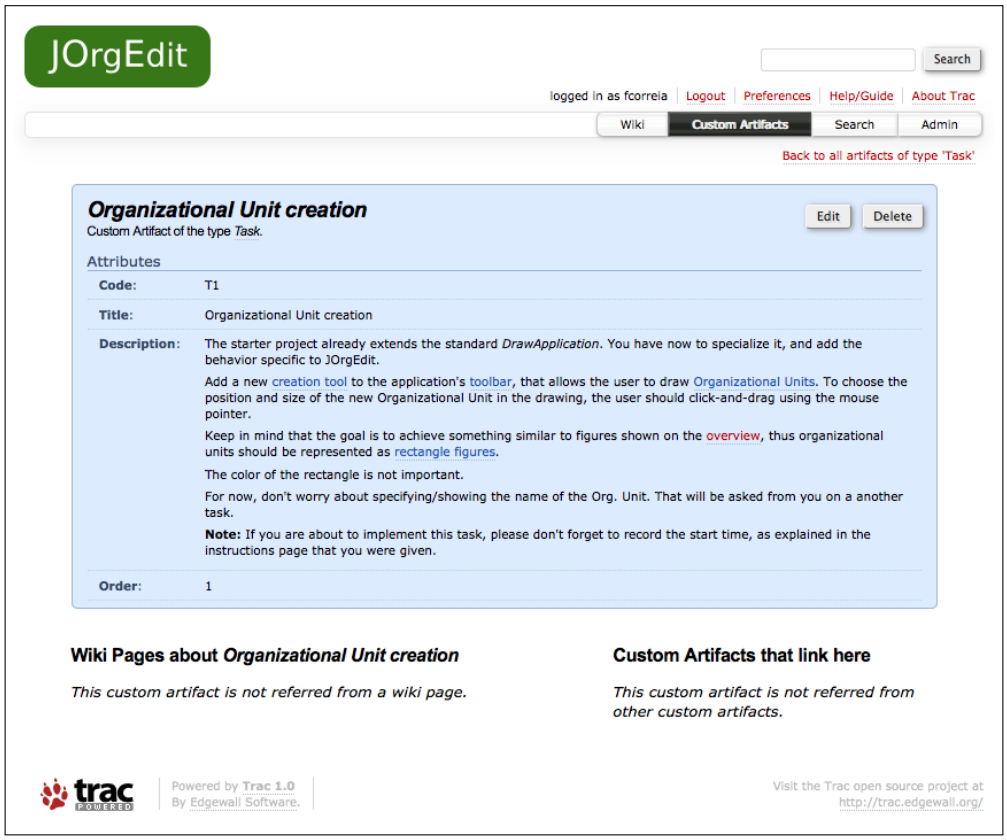


Figure B.7: Page with the description of the first task, part of the platform instructions provided to the Experimental Group.

Assessment Questionnaire

Assessment Questionnaire

Thank you for your participation! We now ask you to answer a few questions about the exercise. It shouldn't take you more than 5 minutes.

Use an 'X' to mark the answers that best reflect your opinions according to the following scale:
1 (Strongly Disagree), 2 (Somewhat Disagree), 3 (Neither Agree nor Disagree), 4 (Somewhat Agree), 5 (Strongly Agree).

External Factors

	1	2	3	4	5
EF1. The room environment was distracting.					
EF2. I found difficulties using the IDE.					
EF3. I found difficulties using the Java language.					

Overall Perception

	1	2	3	4	5
OP1. I found it easy to translate my knowledge of the problem domain to a concrete solution.					
OP2. The project's documentation was easy to use.					
OP3. The tasks descriptions were easy to understand					
OP4. I enjoyed the programming exercise.					

Information Attributes

IA. The information that was made available was ...

	1	2	3	4	5
... in sufficient quantity.					
... not in excessive quantity.					
... of good quality.					
... very precise (i.e., accurate; objective)					
... very concise (i.e, terse; succinct)					

Figure B.8: First page of the assessment questionnaire, used after the tasks, to assess some effects that cannot be measured objectively by the environment.

Classification

	1	2	3	4	5
CL1. I could easily *find* the information that I needed.					
CL2. The way in which the information was organized and linked allowed me to *find* it more easily.					
CL3. I found what I needed to know by browsing the available contents.					
CL4. I found what I needed to know by using Trac's *Search* feature.					

Understandability

	1	2	3	4	5
UN1. The information that was made available was always easy to *understand*.					
UN2. The way in which the information was organized and linked allowed me to *understand* it more easily.					

Consistency

	1	2	3	4	5
CO1. The information that was made available was often inconsistent.					
CO2. I don't have a good perception if the information that was available to me was consistent or not.					

Other

QL1. What worked well?
Please tell us what you think worked well with the documentation contents and the documentation tools that you have used during the assignment.

Figure B.9: Second page of the assessment questionnaire, used after the tasks, to assess some effects that cannot be measured objectively by the environment.

QL2. Improvement suggestions?
Considering the documentation contents and documentation tools that you have used during the assignment, tell us what you think didn't work so well and could be done differently.

QL3. Tell us more!
Use this space to tell us anything that isn't necessarily something that worked well or an improvement. Maybe there was something that you found interesting or intriguing?

F1. Follow-up (optional)

Would you like to be informed of the results of this study by email? Yes [ ] No [ ]

Name: _____

E-mail address: _____

After filling-in the questionnaire, please store it inside the envelope that you have received.

Thank you!

Figure B.10: Third page of the assessment questionnaire, used after the tasks, to assess some effects that cannot be measured objectively by the environment.

Appendix C

Experiment Data

This appendix includes data collected during the statistical experiment described in Chapter 8. The platform activity is not included in this appendix due to its dimension, but can be downloaded from the Web address `http://bit.ly/1iP5Oaj`.

Student Grades

Subject	FPRO	PROG	AEDA
lp00061	19	18	19
lp00062	18	14	14
lp00063	18	14	13
lp00064	18	13	12
lp00065	18	15	12
lp00066	14		
lp00067	13		
lp00068	18	18	18
lp00069	15	15	15
lp00070	19	17	19
lp00071	17	17	17
lp00072	17	18	13
lp00073	15		12
lp00074	11	11	14
lp00075			11

Table C.1: Grades of the subjects of the experimental group for the three courses of their academic track most relevant to the experiment's tasks — FPRO (Programming Fundamentals), PROG (Programming) and AEDA (Algorithms and Data Structures). Empty table cells respect unavailable grades at the time of the data analysis. Subjects are anonymous and are identified by a code.

Subject	FPRO	PROG	AEDA
lp00002	17	16	14
lp00004	17	16	14
lp00005	17	15	14
lp00006	19	17	17
lp00007			10
lp00009	18	15	17
lp00010	14	10	13
lp00014	16	12	13
lp00015	18	18	19
lp00016	19	18	18
lp00018			17
lp00019	16	15	14
lp00020	15	17	17
lp00022	16	17	16
lp00023	17	15	
lp00024	16	15	16
lp00025	13	14	13
lp00027			13
lp00028	18	19	19

Table C.2: Grades of the subjects of the control group for the three courses of their academic track most relevant to the experiment's tasks — FPRO (Programming Fundamentals), PROG (Programming) and AEDA (Algorithms and Data Structures). Empty table cells respect unavailable grades at the time of the data analysis.

Task Durations

Table C.3: Task durations, represented by the times in which each subject has started and finished each task. The *completed* column shows which tasks were effectively completed, and was determined by inspecting the produced source code. Subject are anonymous and identified by a code.

Control Group					Experimental Group				
Subject	Task	Start	End	Completed	Subject	Task	Start	End	Completed
	t1	15:12	15:21	yes		t1	17:40	18:00	yes
	t2	15:21	15:27	yes		t2	18:00	18:04	yes
lp00002	t3	15:27	15:56	yes	lp00061	t3	18:04	18:12	yes
	t4	15:56		no		t4	18:12	18:23	yes
	t5			no		t5	18:23	18:25	yes

continued ...

. . . continued

Control Group					Experimental Group				
Subject	Task	Start	End	Completed	Subject	Task	Start	End	Completed
	t1	15:11	15:41	yes		t1	17:30	17:45	yes
	t2	15:41	15:54	yes		t2	17:45	17:50	yes
lp00004	t3	15:54	16:04	yes	lp00062	t3	17:50	18:03	yes
	t4	16:04		no		t4	18:03	18:25	partial
	t5			no		t5	18:25	18:31	yes
	t1	15:10	15:34	yes		t1	17:40	17:54	yes
	t2	15:34	15:43	yes		t2	17:54	17:57	yes
lp00005	t3	15:43	15:59	yes	lp00063	t3	17:57	18:05	yes
	t4	15:59		no		t4	18:05	18:24	yes
	t5			no		t5	18:24	18:30	yes
	t1	15:10	15:24	yes		t1	18:02	18:13	yes
	t2	15:24	15:37	yes		t2	18:13	18:24	yes
lp00006	t3	15:37	15:50	yes	lp00064	t3	18:24	18:32	yes
	t4	15:50		no		t4	18:32		no
	t5			no		t5			no
	t1	15:55		no		t1	18:00	18:10	yes
	t2			no		t2	18:10	18:15	yes
lp00007	t3			no	lp00065	t3	18:15	18:25	yes
	t4			no		t4	18:25	18:35	partial
	t5			no		t5	18:35		no
	t1	15:15	15:51	yes		t1	17:58		no
	t2	15:51	15:59	yes		t2			no
lp00009	t3	15:59	16:06	yes	lp00066	t3			no
	t4	16:06		no		t4			no
	t5			no		t5			no
	t1	15:33		no		t1	17:40		no
	t2			no		t2			no
lp00010	t3			no	lp00067	t3			no
	t4			no		t4			no
	t5			no		t5			no

continued . . .

... continued

Control Group					Experimental Group				
Subject	Task	Start	End	Completed	Subject	Task	Start	End	Completed
	t1	15:00	15:40	yes		t1	17:39	17:55	yes
	t2	15:40	15:49	yes		t2	17:55	18:00	yes
lp00014	t3	15:49	16:02	yes	lp00068	t3	18:00	18:09	yes
	t4	16:02		no		t4	18:09	18:28	yes
	t5			no		t5	18:28	18:35	yes
	t1	15:19	15:28	yes		t1	17:30	18:00	yes
	t2	15:28	15:35	yes		t2	18:00	18:06	yes
lp00015	t3	15:35	15:42	yes	lp00069	t3	18:06	18:17	yes
	t4	15:42	16:11	yes		t4	18:17	18:38	yes
	t5	16:11	16:14	yes		t5	18:38		no
	t1	15:25	15:36	yes		t1	17:36	17:53	yes
	t2	15:38	15:45	yes		t2	17:53	17:59	yes
lp00016	t3	15:46	16:00	yes	lp00070	t3	17:59	18:12	yes
	t4	16:00	16:10	no		t4	18:12	18:30	yes
	t5	16:10	16:13	yes		t5	18:30	18:34	yes
	t1	15:30	15:36	yes		t1	17:46	17:51	yes
	t2	15:36	15:42	yes		t2	17:51	17:57	yes
lp00018	t3	15:42	15:48	yes	lp00071	t3	17:57	18:08	yes
	t4	15:48	15:58	yes		t4	18:08	18:10	partial
	t5	15:58	16:04	yes		t5	18:10	18:12	yes
	t1	15:20	15:34	yes		t1	17:45	18:00	yes
	t2	15:34	15:39	yes		t2	18:00	18:15	yes
lp00019	t3	15:39	15:43	yes	lp00072	t3	18:15	18:22	yes
	t4	15:43	16:02	partial		t4	18:22	18:26	no
	t5	16:02	16:10	yes		t5	18:26	18:29	yes
	t1	15:16	15:30	yes		t1	17:44	18:08	yes
	t2	15:30	15:33	yes		t2	18:08	18:12	yes
lp00020	t3	15:33	16:03	yes	lp00073	t3	18:12		no
	t4	16:03		no		t4			no
	t5			no		t5			no

continued ...

... continued

Control Group				
Subject	Task	Start	End	Completed
	t1	15:06	15:22	yes
	t2	15:22	15:35	yes
lp00022	t3	15:35	15:47	yes
	t4	15:47	15:53	partial
	t5	15:53	16:00	yes
	t1	15:16	15:28	yes
	t2	15:28	15:34	yes
lp00023	t3	15:34	15:41	yes
	t4	15:41		no
	t5			no
	t1	15:14	15:45	yes
	t2	15:45	15:57	yes
lp00024	t3	15:57	16:03	partial
	t4	16:03	16:16	yes
	t5	16:16	16:23	yes
	t1	15:20	15:45	yes
	t2	15:45	15:50	yes
lp00025	t3	15:50	15:55	yes
	t4	15:55	16:15	yes
	t5	16:15	16:20	yes
	t1	15:00	15:39	yes
	t2	15:39	15:49	yes
lp00027	t3	15:49	16:14	yes
	t4	16:14		no
	t5			no
	t1	15:08	15:16	yes
	t2	15:16	15:21	yes
lp00028	t3	15:21	15:30	yes
	t4	15:30	15:44	yes
	t5	15:44	15:49	yes

Experimental Group				
Subject	Task	Start	End	Completed
	t1	17:43	18:15	yes
	t2	18:15		no
lp00074	t3			no
	t4			no
	t5			no
	t1	17:25	17:40	yes
	t2	17:40	17:50	yes
lp00075	t3	17:50	18:10	yes
	t4	18:10		no
	t5			no

Questionnaires Answers

	Control Group																			Experimental Group																		
BG1.1	4	4	3	4	5	3	3	4	3	5	3	3	4	4	4	4	4	4	4	4	5	5	3	4	3	4	5	4	4	4	5	4	3	4
BG1.2	4	4	4	4	5	4	4	4	3	5	4	5	4	4	4	5	5	5	3	4	5	5	4	4	3	4	5	5	4	4	4	4	4	4
BG1.3	1	3	3	5	1	1	2	1	3	1	4	2	4	3	1	1	3	2	4	4	3	3	1	3	2	3	4	3	2	2	2	1	1	3
BG1.4	3	1	1	5	1	1	1	1	1	1	1	1	1	1	1	1	3	1	3	1	2	2	1	1	2	3	2	2	1	1	2	1	1	1
BG1.5	1	1	2	5	1	1	1	1	1	1	1	1	1	1	1	1	3	1	3	1	2	2	1	1	1	2	1	1	1	1	2	1	1	1
BG1.6	2	1	1	3	3	1	1	1	3	3	4	3	1	2	1	3	4	3	4	2	4	3	1	2	1	4	2	3	3	1	2	1	1	2
BG1.7	3	1	1	3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	1	1	1	3	1	1	1	1	2	1	1	1
BG1.8	4	4	4	5	4	4	3	5	5	5	5	4	4	5	4	5	5	4	4	4	4	5	4	4	4	4	4	4	4	4	4	4	3	3
BG1.9	2	2	4	5	4	4	3	4	4	5	5	5	4	4	1	4	5	3	4	3	2	5	4	4	2	4	3	5	3	3	4	4	1	4
BG1.10	1	1	2	5	2	1	1	1	1	3	3	1	2	3	1	4	1	3	3	1	2	3	1	1	1	4	1	4	2	1	4	2	1	1
BG1.11	1	1	2	5	2	1	1	1	1	3	2	1	2	2	1	4	1	4	4	1	2	2	2	1	1	3	1	3	2	1	4	2	1	1
BG1.12	2	2	4	5	2	2	3	4	3	4	3	3	3	3	4	3	4	5	3	2	3	4	2	1	3	4	2	4	4	3	3	4	1	3
BG1.13	2	1	3	5	3	3	2	3	4	3	2	1	4	2	3	3	4	4	3	3	3	3	3	1	3	3	2	4	4	3	3	3	1	3
BG1.14	3	2	2	4	3	4	1	3	3	2	3	1	3	2	3	3	3	4	4	4	5	4	3	1	1	4	2	4	4	3	2	4	1	3
BG1.15	2	4	3	4	3	3	3	4	2	4	2	4	3	2	3	4	4	3	3	5	3	5	2	2	4	4	4	4	3	3	4	2	2	3
EF1	2	1	1	1	2	1	2	1	1	3	2	2	2	2	1	1	1	2	1	1	2	1	2	1	1	3	1	1	1	1	1	1	1	1
EF2	1	4	1	1	2	1	3	1	1	2	1	1	1	1	1	1	1	2	1	1	2	1	1	3	3	3	4	1	1	1	3	1	2	1
EF3	2	3	1	1	2	1	4	1	1	1	1	2	1	1	1	1	1	1	1	1	2	1	1	3	3	1	1	1	1	1	1	3	2	
OP1	4	2	3	3	2	3	1	3	5	4	3	3	4	4	3	4	4	3	3	3	4	5	3	5	2	1	4	4	4	4	3	3	1	5
OP2	3	1	3	3	3	4	3	3	3	4	3	5	3	5	4	2	2	4	5	3	3	4	2	5	2	1	2	5	5	5	2	2	2	4
OP3	4	2	4	3	4	2	3	4	4	4	5	3	4	3	5	4	4	5	5	5	3	4	3	5	3	1	5	4	4	4	2	2	2	4
OP4	2	3	4	2	4	2	3	4	5	4	5	5	4	4	4	5	5	4	4	5	4	5	4	5	4	1	3	5	5	5	4	4	1	4
IA1	4	3	3	4	3	2	3	5	3	4	3	5	4	4	4	5	5	5	5	5	5	5	3	5	1	1	4	5	4	5	4	2	2	5
IA2	4	2	3	3	4	2	2	4	5	4	2	4	3	5	5	2	4	4	2	2	3	4	3	5	2	1	2	4	5	5	4	4	4	4
IA3	4	1	4	3	4	3	4	4	4	4	4	5	3	3	4	4	5	5	4	5	4	4	3	5	3	2	2	5	4	5	4	4	3	4
IA4	4	2	4	4	4	3	4	3	2	4	4	3	3	3	4	3	5	3	3	4	3	4	3	5	2	1	2	5	4	5	3	4	3	4
IA5	3	2	4	4	2	3	3	3	3	4	4	2	3	2	4	3	4	3	3	5	4	4	2	5	2	1	4	5	4	5	3	4	3	5
CL1	2	2	2	4	2	4	2	3	3	2	5	4	2	4	4	2	4	3	4	4	2	4	2	5	2	2	2	5	4	4	3	2	2	4
CL2	4	4	4	3	3	3	3	4	2	4	2	3	4	2	3	3	4	4	5	4	3	4	2	5	2	2	1	5	4	5	3	4	3	4
CL3	3	2	3	4	1	2	3	1	5	3	1	4	4	3	3	4	3	3	5	5	4	4	4	5	2	2	4	4	4	5	3	4	1	5
CL4	3	3	4	4	2	5	3	5	2	1	5	3	4	5	4	3	3	1	1	1	3	4	3	3	3	2	1	3	3	1	3	3	3	4
UN1	3	1	4	4	2	3	4	3	3	3	4	3	3	4	4	3	2	4	5	4	4	4	4	5	3	2	4	4	4	4	3	5	3	4
UN2	4	4	4	4	3	3	3	3	4	3	3	4	4	5	4	3	4	5	4	4	3	5	3	5	3	3	3	5	3	5	4	5	3	4
CO1	2	4	2	2	3	4	2	2	1	2	1	1	3	3	1	3	4	2	3	1	1	2	3	1	2	3	2	2	1	1	2	3	1	2
CO2	2	5	1	3	2	4	2	2	3	2	1	3	3	4	2	2	1	1	2	3	3	1	4	1	4	3	3	1	2	1	2	3	4	1

Table C.4: Answers to the background and assessment questionnaires. Each row represents a different questionnaire item, and each column the answers of a specific subject.

Appendix D

ASA Analyzer

The *ASA Analyzer* module was developed using the Python programming language and uses the `numpy` and `matplotlib` modules respectively to run the statistical tests and plot the bar-charts and boxplots. The module takes two main responsibilities – the random assignment of subjects to the experimental groups and the analysis of data collected during the experiment. The latest version may be obtained from the Web address https://github.com/filipefigcorreia/asaanalyzer.

Random Assignment

The module uses python's pseudo-random number generator to distribute subjects among two experimental groups, and validates the resulting assignment by comparing the grades of the two groups on a set of courses using an independent-samples Mann-Whitney U test. Data about the students and their grades is loaded from a spreadsheet file and the output is a) a comma-separated values (CSV) file with the group assignment, b) a boxplot diagram of the grades and c) latex files with the results of the statistical tests, that were later used to include the results in this dissertation.

Analysis of Collected Data

The module loads data collected during the experiment and stored in the local filesystem as spreadsheet and database files. It then produces CSV files with results of statistical tests, PDF files of charts of the data used in its raw or aggregated forms, and latex files for inclusion of the results in this dissertation. More specifically, the analyses

run by this module focus on a) the answers to the background questionnaire, b) the platform activity, c) the task times and d) the answers to the assessment questionnaire.

At its core, the ASAAnalyzer module is composed by the classes depicted in Figure D.1. The `ASAExperiment` class represents an experiment, with its several experimental groups (`ASAExperimentGroup`) and calculations resulting from a statistical analysis (`ASAExperimentCalculations`). The `run_tests()` method processes the data collected for each experimental group (stored in memory as instances of the `ASADataSet` class) by running the statistical tests and it creates an instance of `ASAExperimentCalculations` that encapsulates all the results (that are also instances of the `ASADataSet` class). The `ASADBDataSet` class represents data that has been loaded directly from a database (i.e., platform activity data, in this case) and provides some behavior specific for that context.

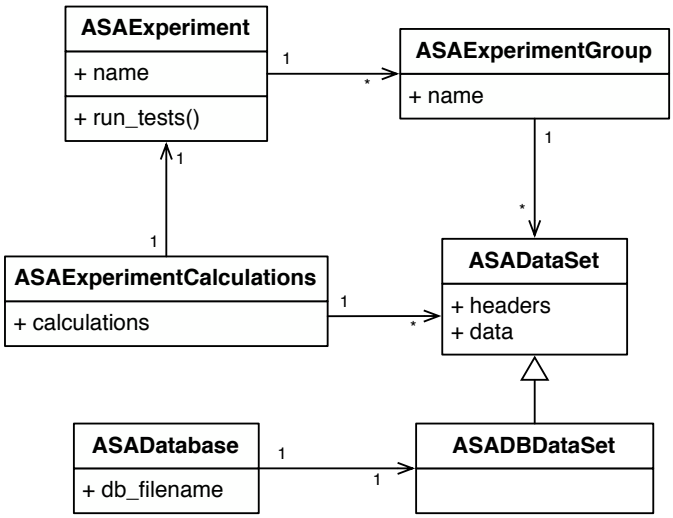


Figure D.1: Core classes of the ASA Analyzer domain model.

Appendix E

Experiment Data Analysis

This appendix includes intermediate numeric results and charts produced during the experiments' data analysis that are not sufficiently relevant for inclusion in Section 8.4 but may still be useful to anyone taking a closer look to the collected data.

Some of the charts included in this appendix are boxplots, which conveniently allow to depict the data highlighting its median and quartiles. *Outliers* are represented as blue crosses outside the range of the *whiskers*. Sometimes it proved useful to show the data points used to plot the chart and they are represented as green dots.

Student Grades

Figure E.1 provides an additional level of detail and shows a boxplot of the data included in Tables C.1 and C.2.

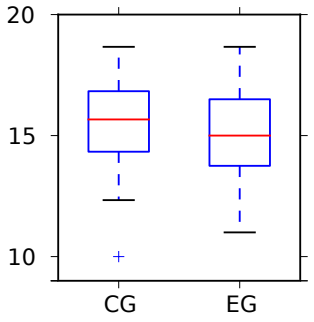


Figure E.1: Boxplot of the subjects' mean grades for a selected set of relevant courses by group. The vertical axis indicates the grades and the horizontal axis indicates the two different groups – the Control Group (CG) and the Experimental Group (EG).

One of the requirements of using the Mann-Whitney U test to compare the grades of the two experiment groups, as was shown in Table 8.2, is that an equal variance

can be assumed. We have used the data included in Tables C.1 and C.2 to assess the equality of variances through a levene test and the results are shown in Table E.1.

W	ρ
0.010	0.920

Table E.1: Result of the levene test for the equality of variances of the students' grades. We can assume an equal variance as the ρ-value is greater than 0.05.

Platform Activity

Figure E.2 provides a view over the data that is similar to the one provided by Figure 8.2 but introduces additional detail by using a boxplot. Figure E.3 uses again the same data but doesn't aggregate it by the Trac modules – Wiki Pages, Search, Adaptive Artifacts and the Adaptive Artifacts' index – thus showing the total durations for each

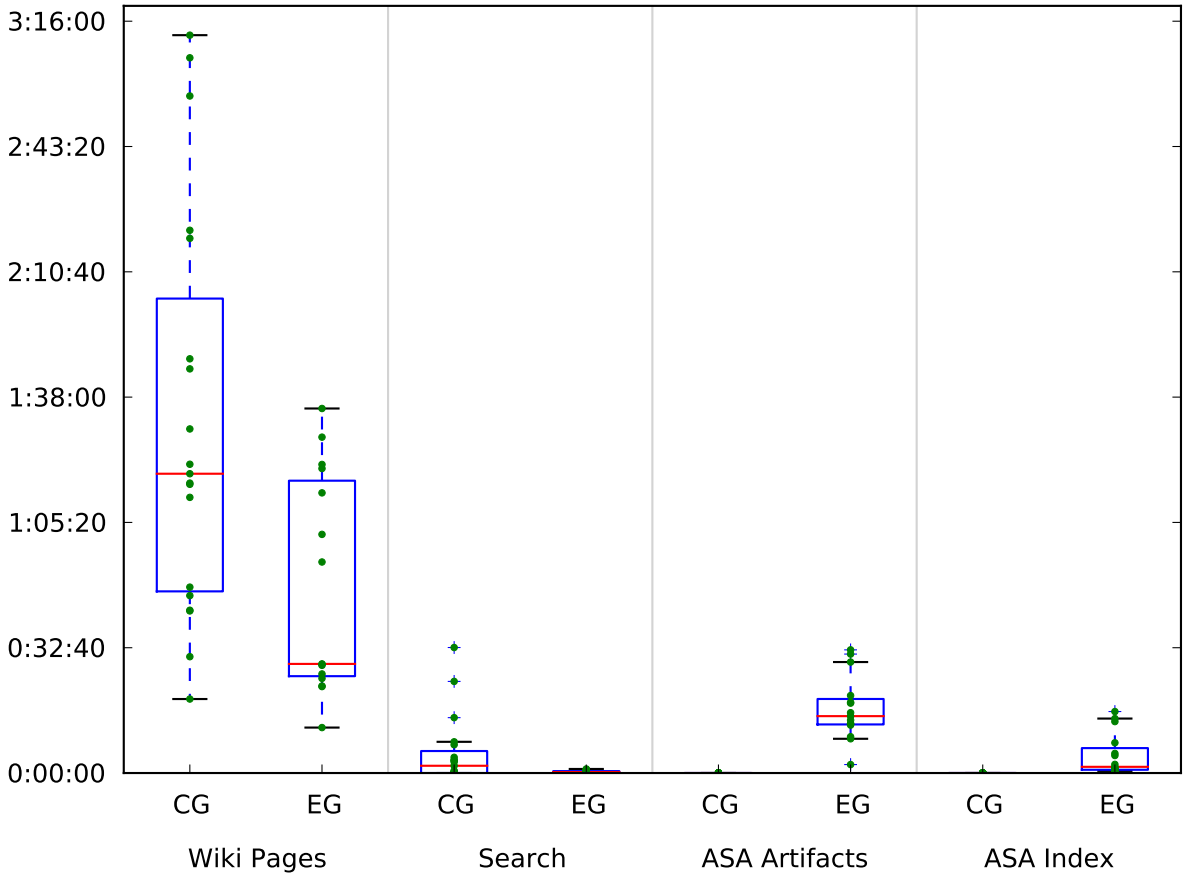


Figure E.2: Boxplot of the mean times spent on the platform by Trac module.

of the groups. The green dots represent data points, that is, the mean times spent by a subject of a specific group – the Control Group (CG) or the Experimental Group (EG).

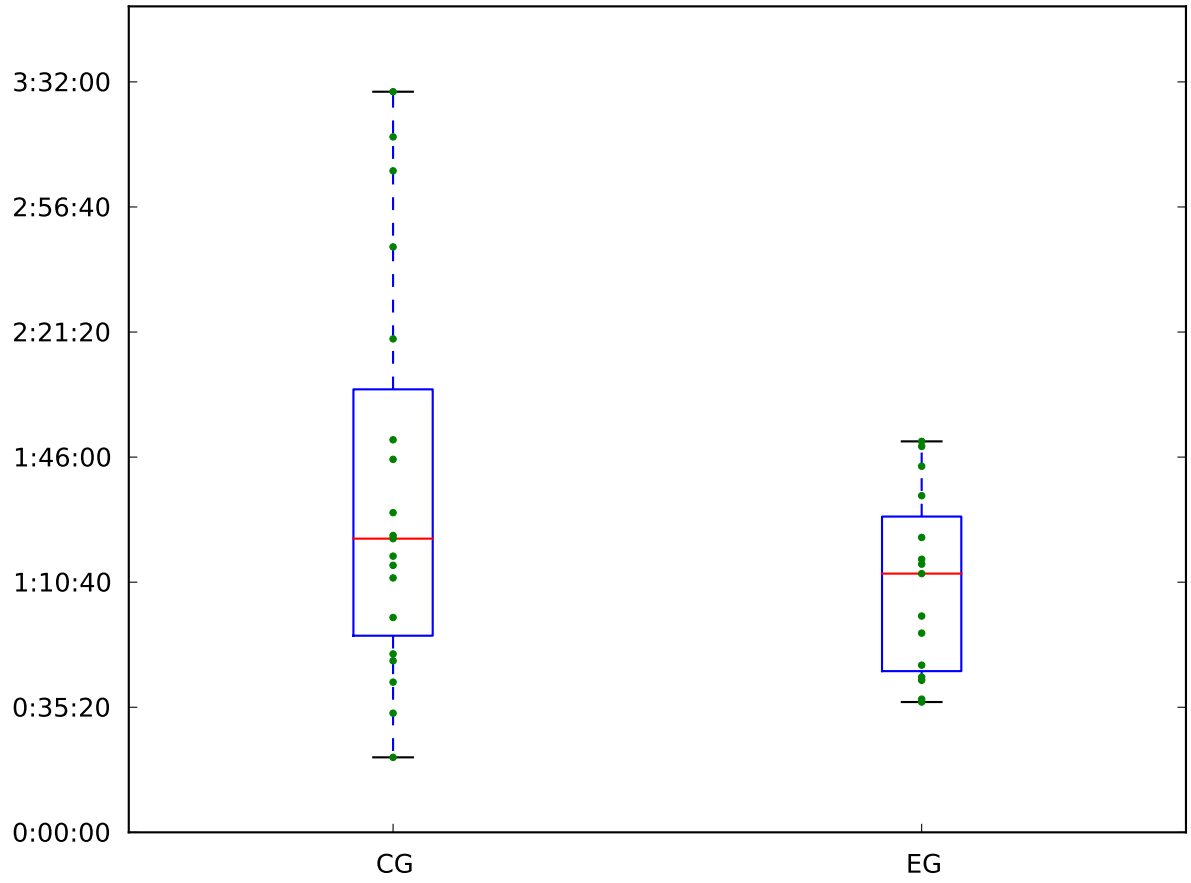


Figure E.3: Boxplot of the mean times spent on the platform.

To use the Mann-Whitney U test to compare the answers to the questionnaire items, as we have shown in Tables 8.3 and 8.14 to 8.19, we have to ensure that an equal variance can be assumed. We have used the data included in Table C.4 to assess the equality of variances through a levene test and the results are shown in Table E.2.

	W	ρ
BG1.1	0.067	0.798
BG1.2	0.259	0.614
BG1.3	1.374	0.250
BG1.4	0.000	0.983
BG1.5	0.486	0.491
BG1.6	0.526	0.474
BG1.7	0.324	0.573
BG1.8	0.655	0.424
BG1.9	0.178	0.676
BG1.10	0.041	0.840
BG1.11	1.126	0.296
BG1.12	0.248	0.622
BG1.13	1.124	0.297
BG1.14	3.307	0.078
BG1.15	2.142	0.153
EF1	1.550	0.222
EF2	1.878	0.180
EF3	0.151	0.700
OP1	1.624	0.212
OP2	3.182	0.084
OP3	1.783	0.191
OP4	0.375	0.545
IA1	3.182	0.084
IA2	0.002	0.964
IA3	0.651	0.426
IA4	1.407	0.244
IA5	2.836	0.102
CL1	1.179	0.286
CL2	1.477	0.233
CL3	0.019	0.891
CL4	2.437	0.128
UN1	0.917	0.345
UN2	4.002	0.054
CO1	0.755	0.391
CO2	0.513	0.479

Table E.2: Result of the levene test for the equality of variances of the answers to the questionnaire items. We can assume equal variances as the ρ-value is greater than 0.05 for all questionnaire items.

Task Durations

The same data represented in Figure 8.3 is represented in Figure E.4 in finer detail, as a boxplot – it shows the mean durations that each subject took to complete each one of the tasks. Figure E.5 shows the mean times that each subject took to complete the *totality* of the tasks.

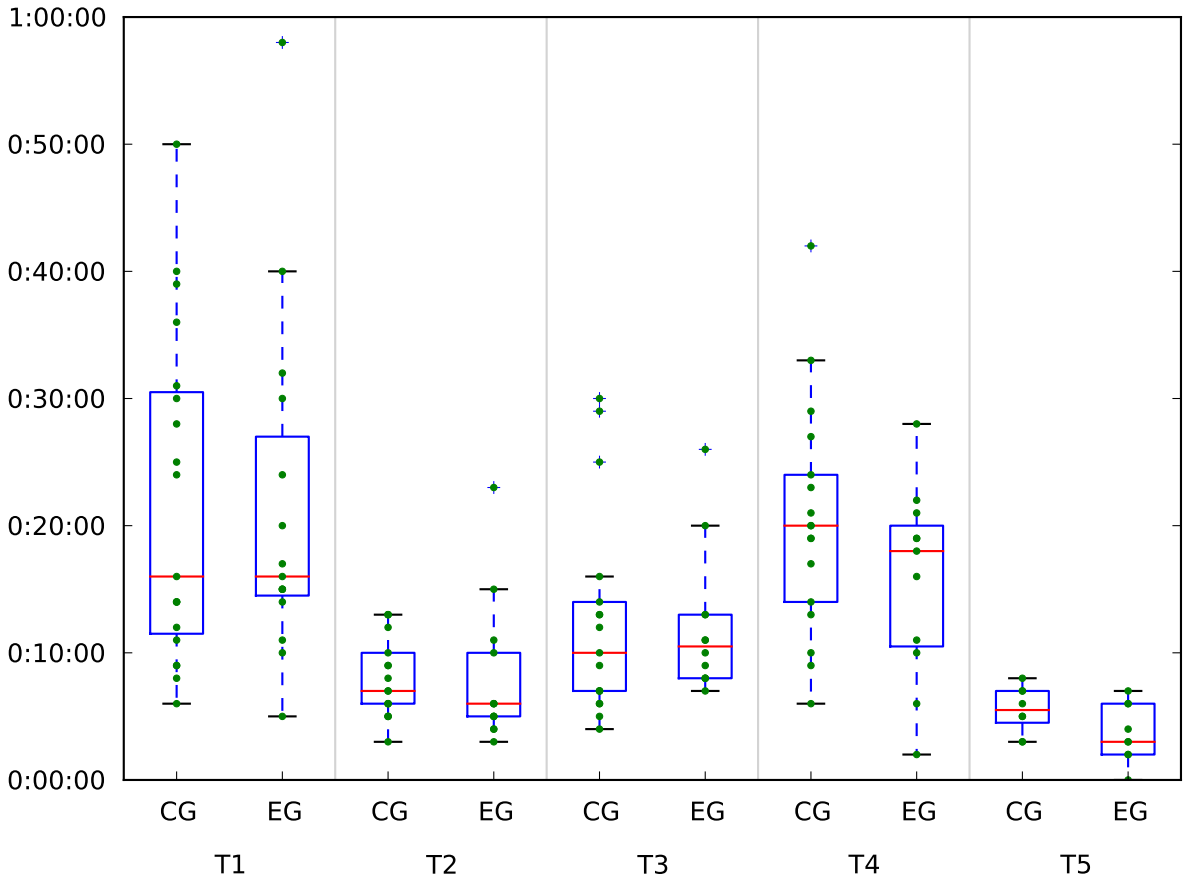


Figure E.4: Boxplot of the mean duration of each task.

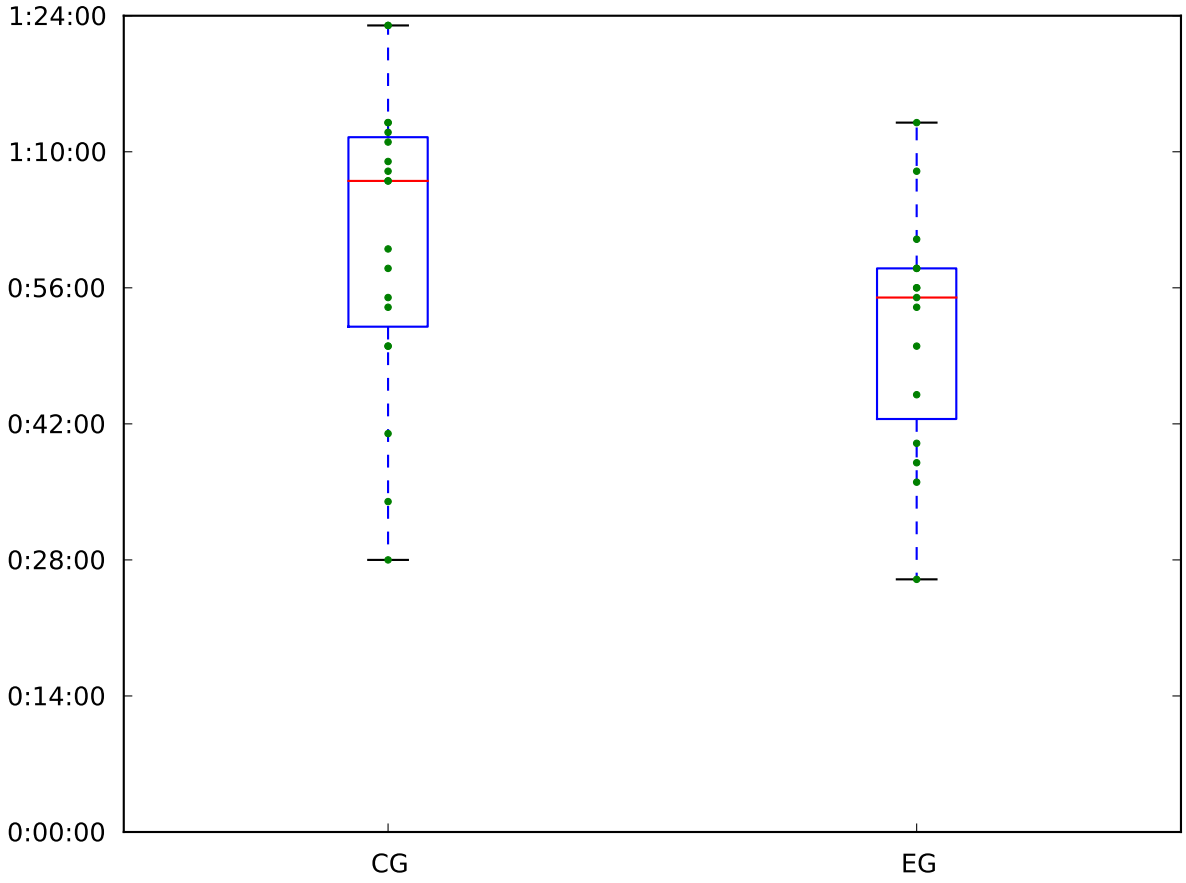
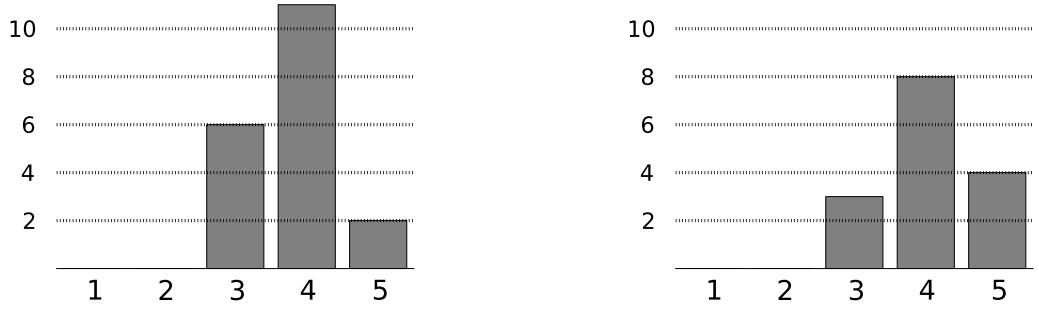


Figure E.5: Boxplot of the mean duration of the totality of the tasks.

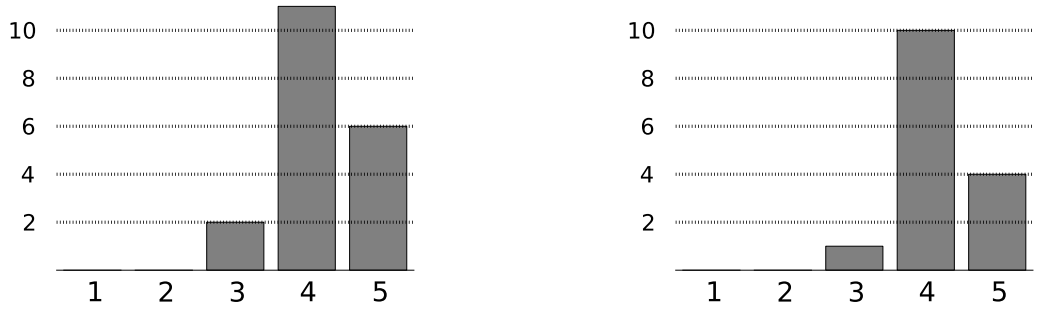
Questionnaire Answers

Figures E.6 to E.40 depict the data of Table C.4. They represent the answers to the items of the background and assessment questionnaires as histograms, as has already been presented in a smaller scale in Tables 8.3 and 8.14 to 8.19.



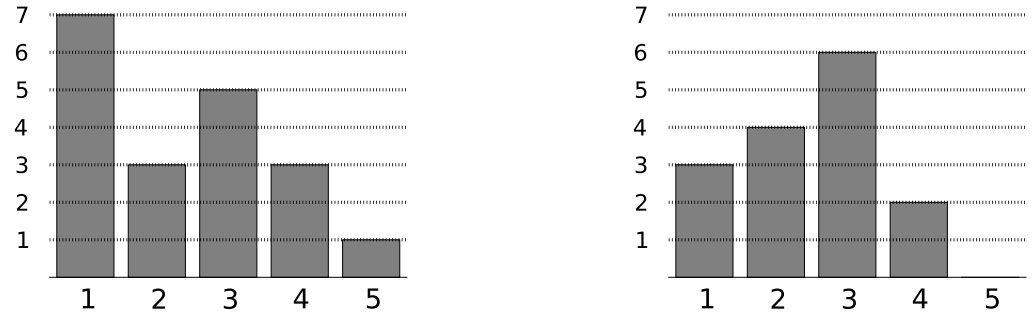
(a) Control Group **(b)** Experimental Group

Figure E.6: Histogram of the answers to the questionnaire item BG1.1 – *I have considerable experience using the Java programming language.*



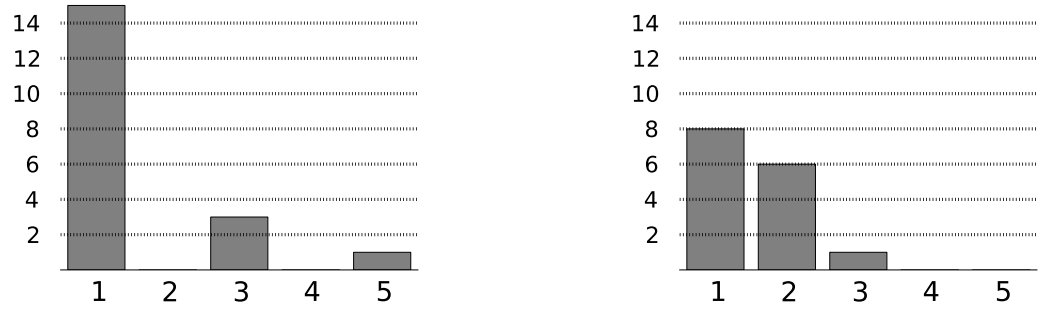
(a) Control Group **(b)** Experimental Group

Figure E.7: Histogram of the answers to the questionnaire item BG1.2 – *I have considerable experience using the Eclipse IDE.*



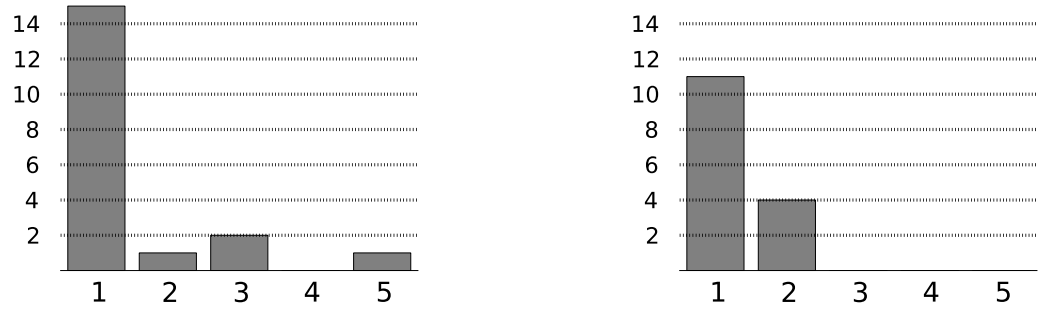
(a) Control Group **(b)** Experimental Group

Figure E.8: Histogram of the answers to the questionnaire item BG1.3 – *I have considerable experience using Software Forges.*



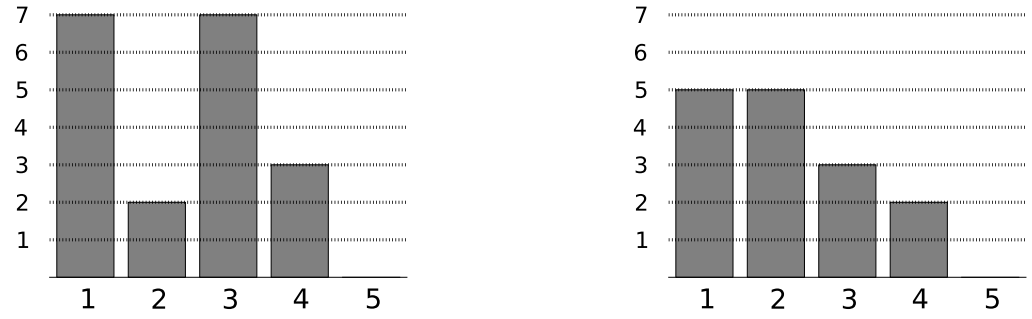
(a) Control Group **(b)** Experimental Group

Figure E.9: Histogram of the answers to the questionnaire item BG1.4 – *I have considerable experience using the Trac platform.*



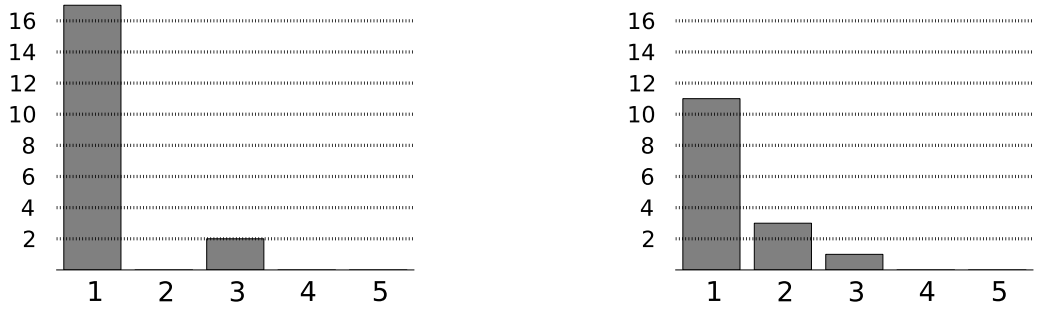
(a) Control Group **(b)** Experimental Group

Figure E.10: Histogram of the answers to the questionnaire item BG1.5 – *I have considerable experience using Trac's Adaptive Software Artifacts or Custom Software Artifacts.*



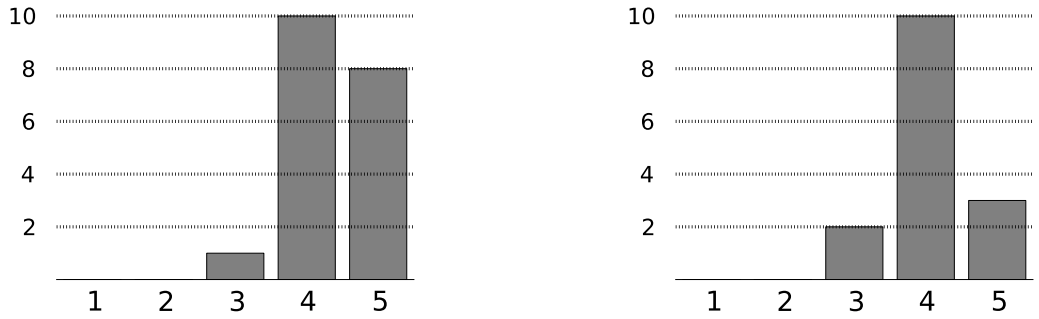
(a) Control Group **(b)** Experimental Group

Figure E.11: Histogram of the answers to the questionnaire item BG1.6 – *I have considerable experience using frameworks.*



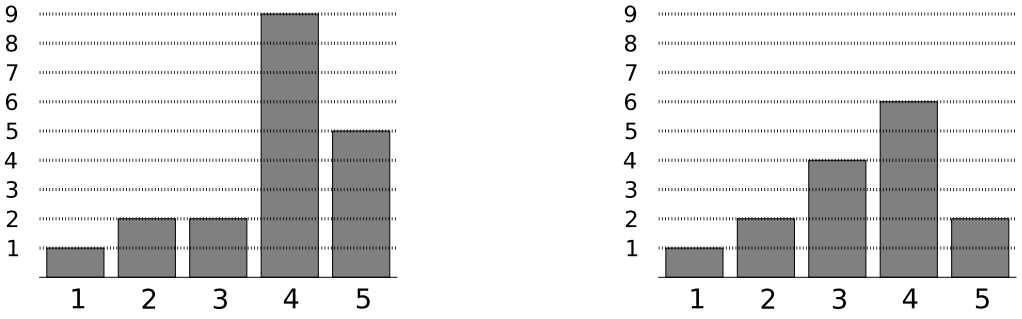
(a) Control Group **(b)** Experimental Group

Figure E.12: Histogram of the answers to the questionnaire item BG1.7 – *I have considerable experience using the JHotDraw framework.*



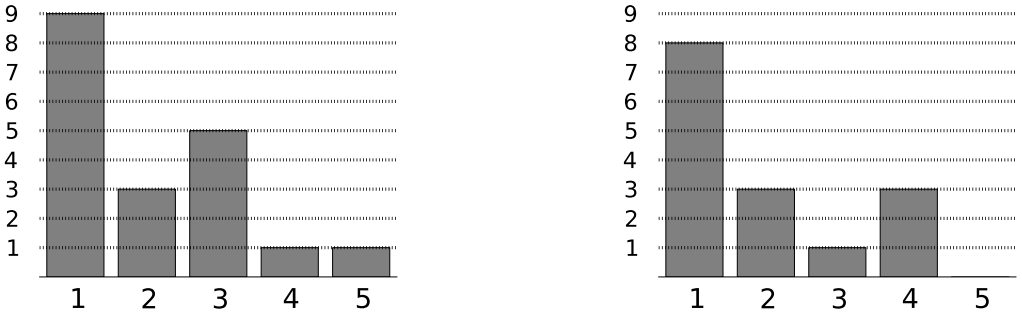
(a) Control Group **(b)** Experimental Group

Figure E.13: Histogram of the answers to the questionnaire item BG1.8 – *I have considerable experience with object-oriented software development.*



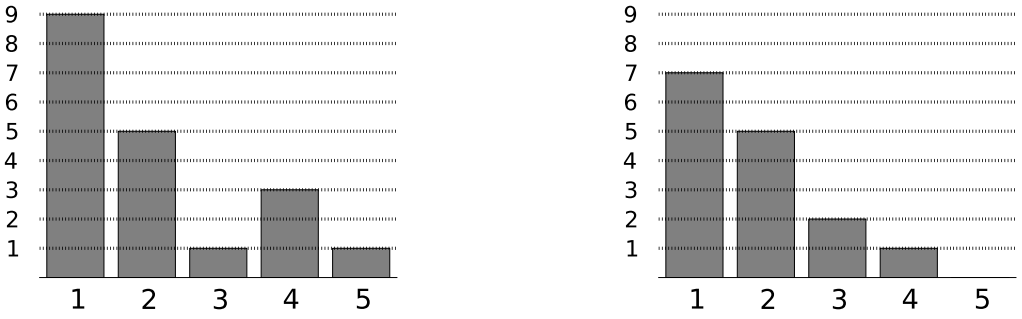
(a) Control Group **(b)** Experimental Group

Figure E.14: Histogram of the answers to the questionnaire item BG1.9 – *I have considerable experience extending a system using composition and subclassing.*



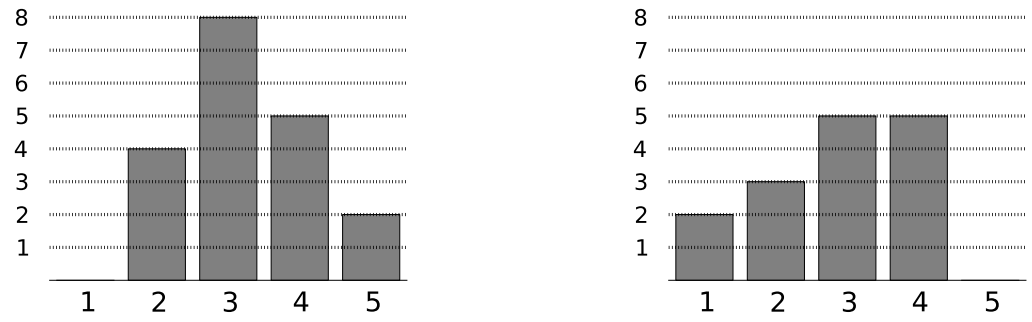
(a) Control Group **(b)** Experimental Group

Figure E.15: Histogram of the answers to the questionnaire item BG1.10 – *I have considerable experience developing industry-level applications.*



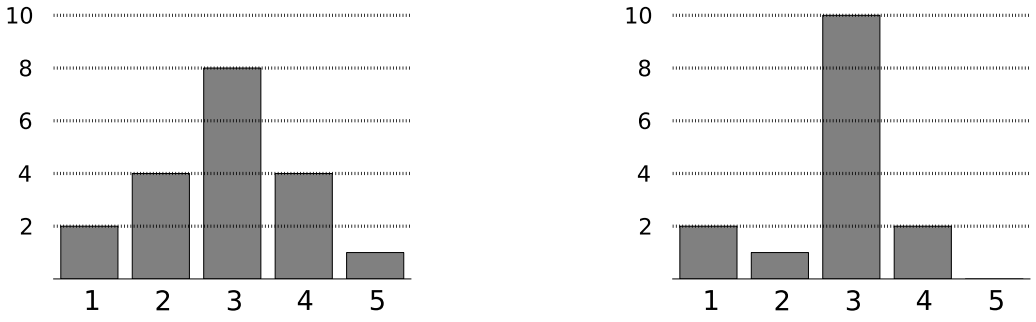
(a) Control Group **(b)** Experimental Group

Figure E.16: Histogram of the answers to the questionnaire item BG1.11 – *I have considerable experience maintaining/modifying industry-level applications.*



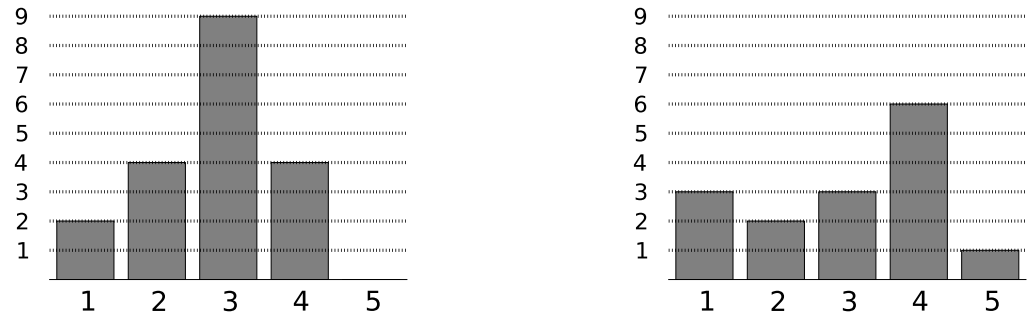
(a) Control Group **(b)** Experimental Group

Figure E.17: Histogram of the answers to the questionnaire item BG1.12 – *I have considerable experience documenting software systems.*



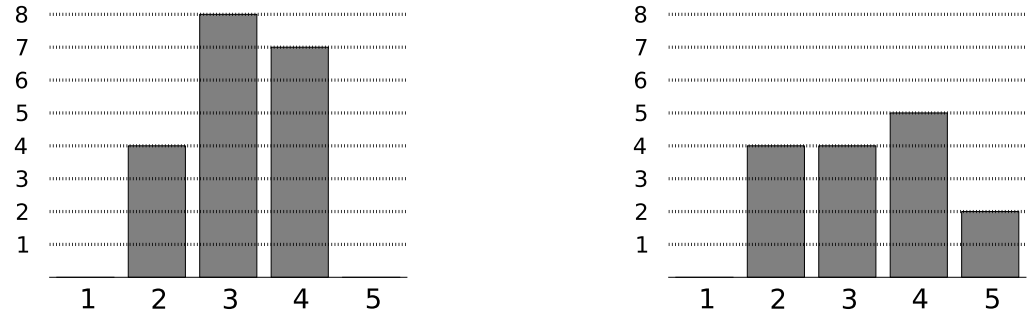
(a) Control Group **(b)** Experimental Group

Figure E.18: Histogram of the answers to the questionnaire item BG1.13 – *I have considerable experience using technical documentation of software systems.*



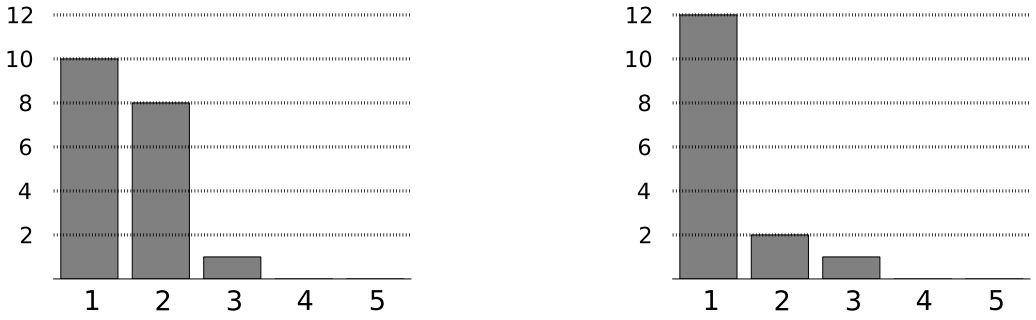
(a) Control Group **(b)** Experimental Group

Figure E.19: Histogram of the answers to the questionnaire item BG1.14 – *I have considerable experience using wikis.*



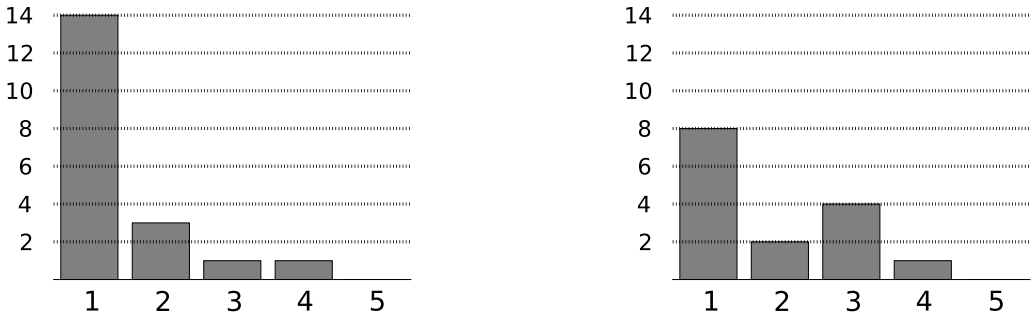
(a) Control Group **(b)** Experimental Group

Figure E.20: Histogram of the answers to the questionnaire item BG1.15 – *I have considerable experience developing standalone GUI (Graphical User Interface) applications.*



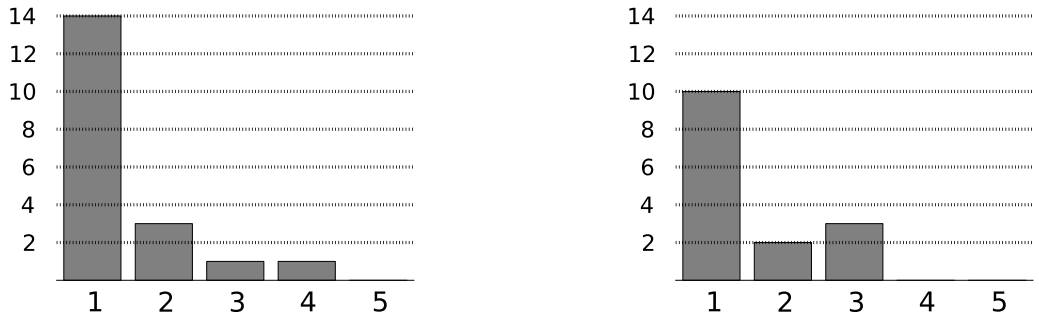
(a) Control Group **(b)** Experimental Group

Figure E.21: Histogram of the answers to the questionnaire item EF1 – *The room environment was distracting.*



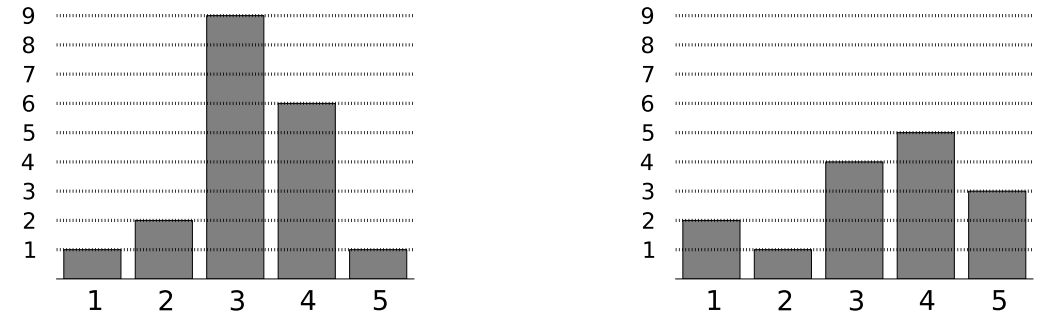
(a) Control Group **(b)** Experimental Group

Figure E.22: Histogram of the answers to the questionnaire item EF2 – *I found difficulties using the IDE.*



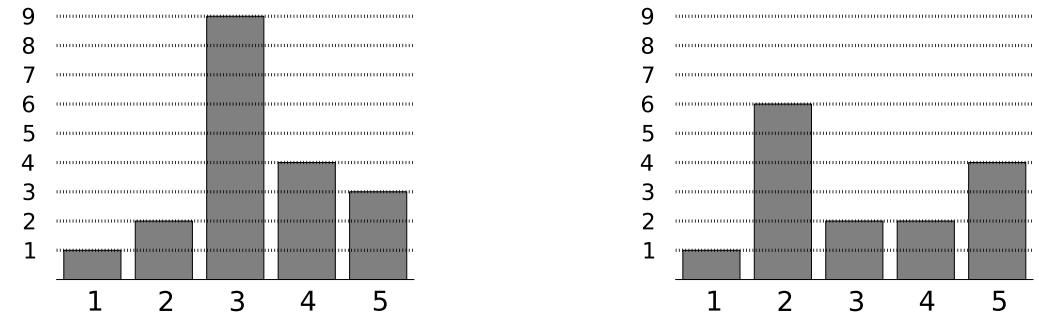
(a) Control Group **(b)** Experimental Group

Figure E.23: Histogram of the answers to the questionnaire item EF3 – *I found difficulties using the Java language.*



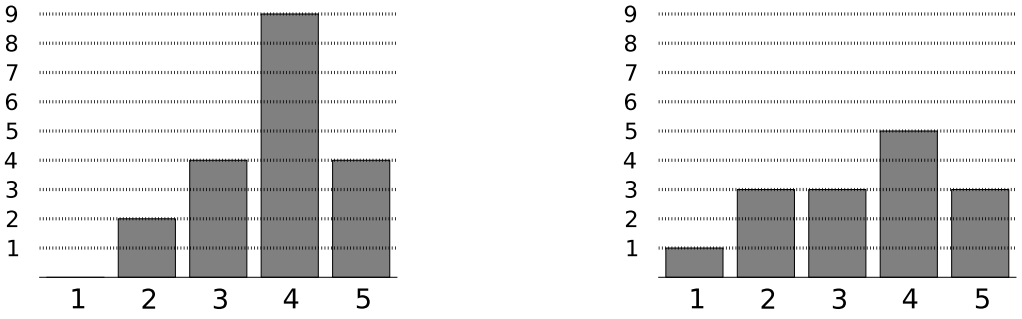
(a) Control Group **(b)** Experimental Group

Figure E.24: Histogram of the answers to the questionnaire item OP1 – *I found it easy to translate my knowledge of the problem domain to a concrete solution.*



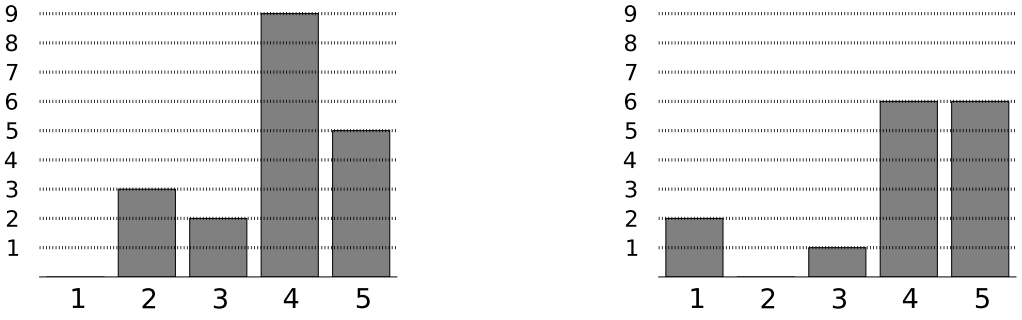
(a) Control Group **(b)** Experimental Group

Figure E.25: Histogram of the answers to the questionnaire item OP2 – *The project's documentation was easy to use.*



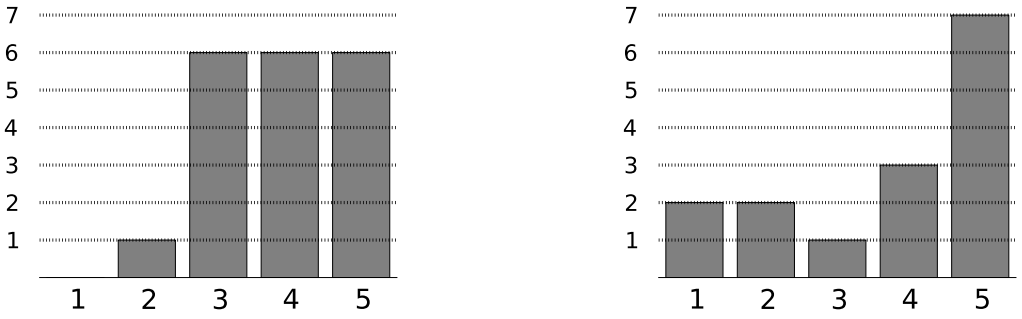
(a) Control Group **(b)** Experimental Group

Figure E.26: Histogram of the answers to the questionnaire item OP3 – *The tasks descriptions were easy to understand.*



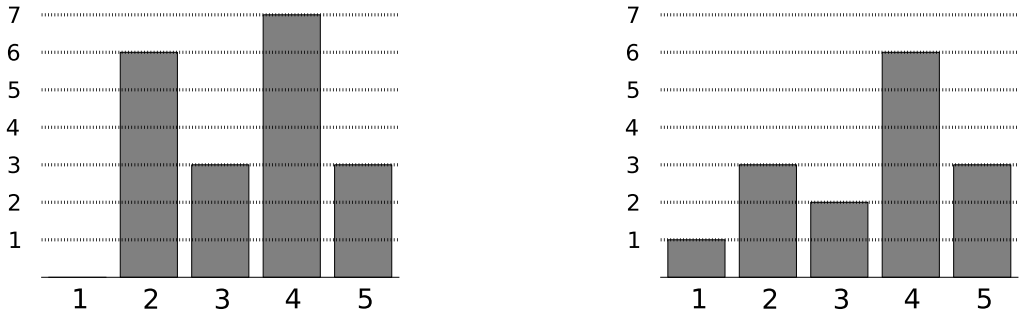
(a) Control Group **(b)** Experimental Group

Figure E.27: Histogram of the answers to the questionnaire item OP4 – *I enjoyed the programming exercise.*



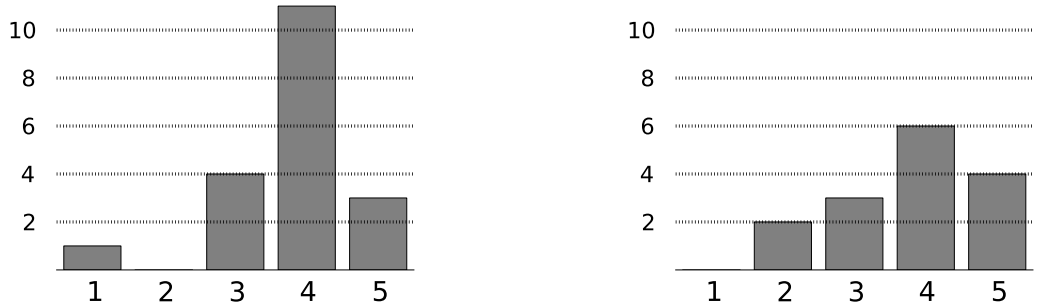
(a) Control Group **(b)** Experimental Group

Figure E.28: Histogram of the answers to the questionnaire item IA1 – *The information that was made available was in sufficient quantity.*



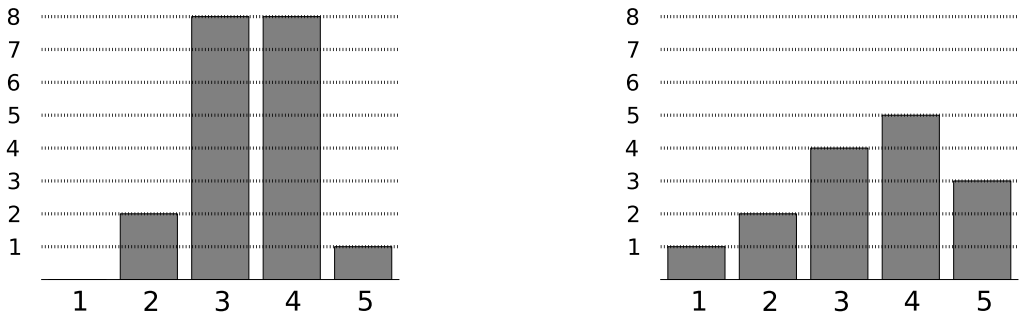
(a) Control Group **(b)** Experimental Group

Figure E.29: Histogram of the answers to the questionnaire item IA2 – *The information that was made available was not in excessive quantity.*



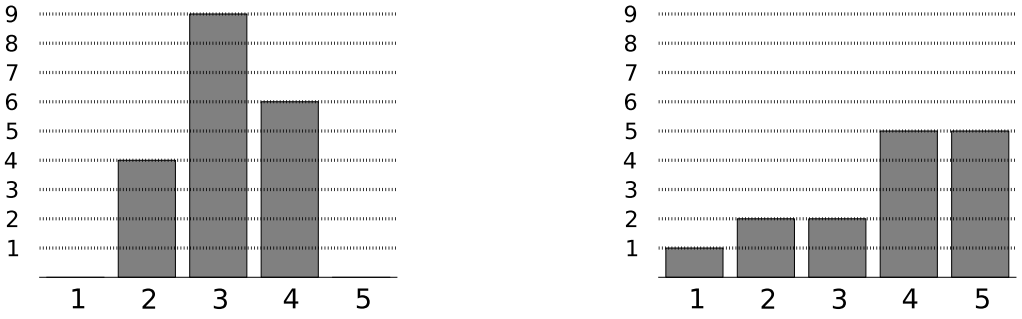
(a) Control Group **(b)** Experimental Group

Figure E.30: Histogram of the answers to the questionnaire item IA3 – *The information that was made available was of good quality.*



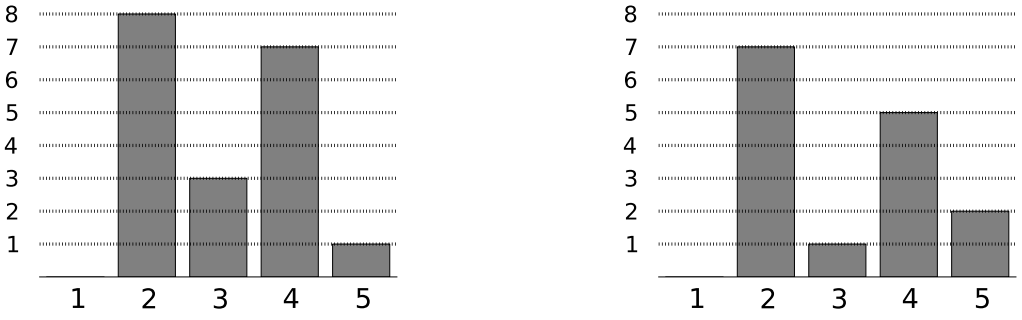
(a) Control Group **(b)** Experimental Group

Figure E.31: Histogram of the answers to the questionnaire item IA4 – *The information that was made available was very precise (i.e., accurate; objective)*



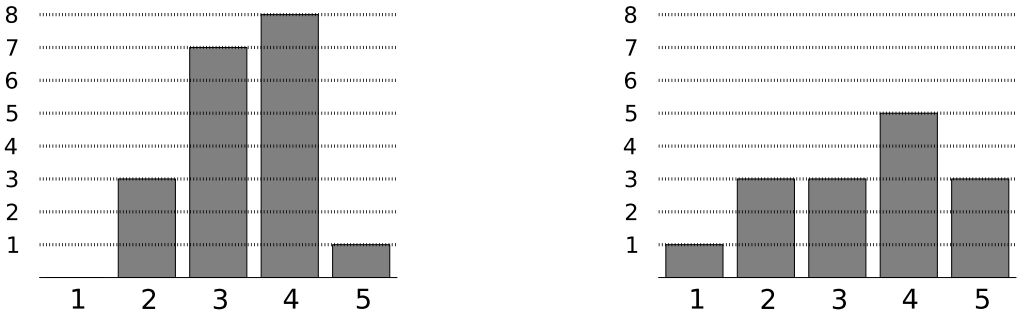
(a) Control Group **(b)** Experimental Group

Figure E.32: Histogram of the answers to the questionnaire item IA5 – *The information that was made available was very concise (i.e, terse; succinct)*



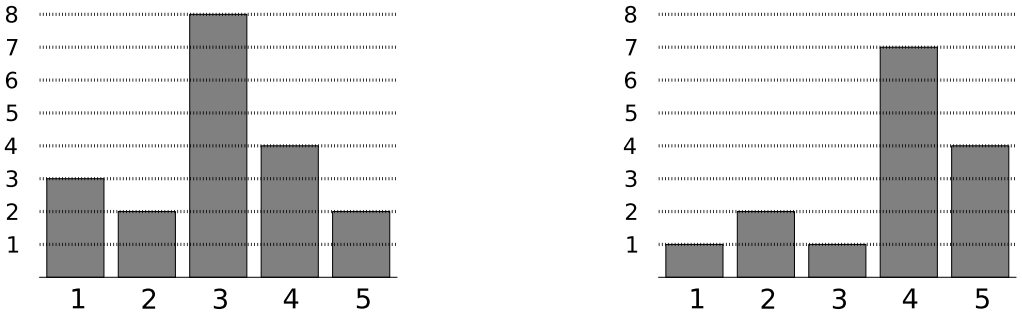
(a) Control Group **(b)** Experimental Group

Figure E.33: Histogram of the answers to the questionnaire item CL1 – *I could easily find the information that I needed.*



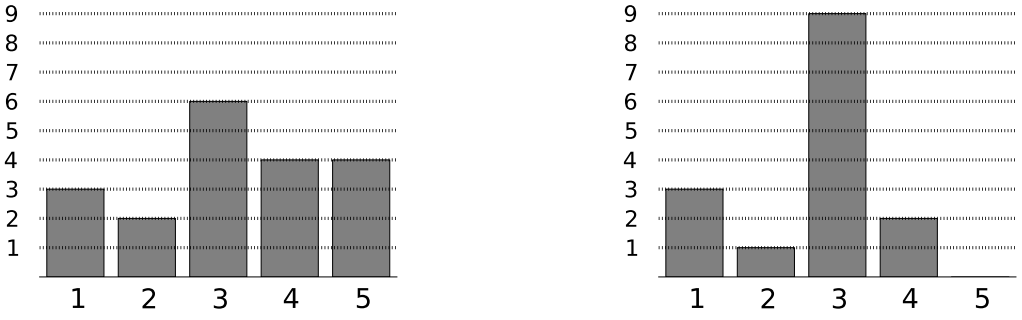
(a) Control Group **(b)** Experimental Group

Figure E.34: Histogram of the answers to the questionnaire item CL2 – *The way in which the information was organized and linked allowed me to find it more easily.*



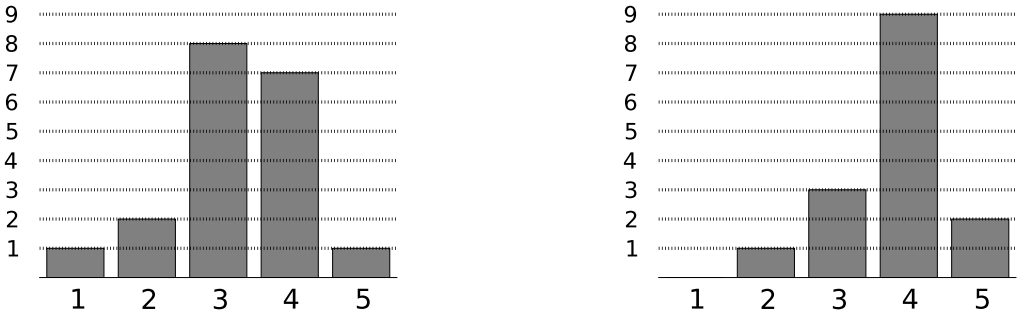
(a) Control Group **(b)** Experimental Group

Figure E.35: Histogram of the answers to the questionnaire item CL3 – *I found what I needed to know by browsing the available contents.*



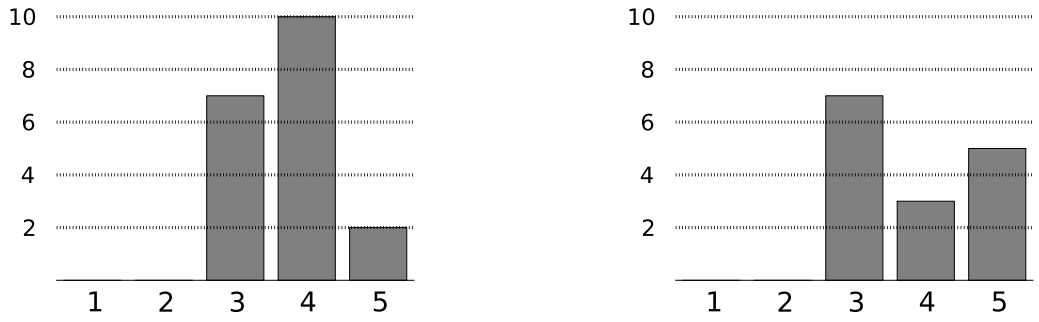
(a) Control Group **(b)** Experimental Group

Figure E.36: Histogram of the answers to the questionnaire item CL4 – *I found what I needed to know by using Trac's Search feature.*



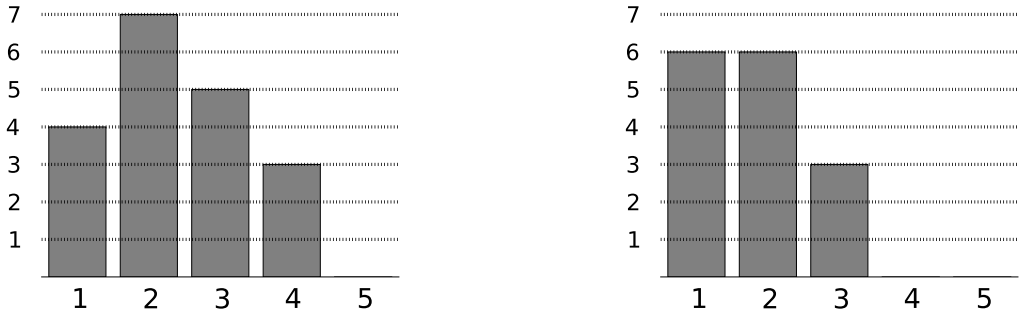
(a) Control Group **(b)** Experimental Group

Figure E.37: Histogram of the answers to the questionnaire item UN1 – *The information that was made available was always easy to understand.*



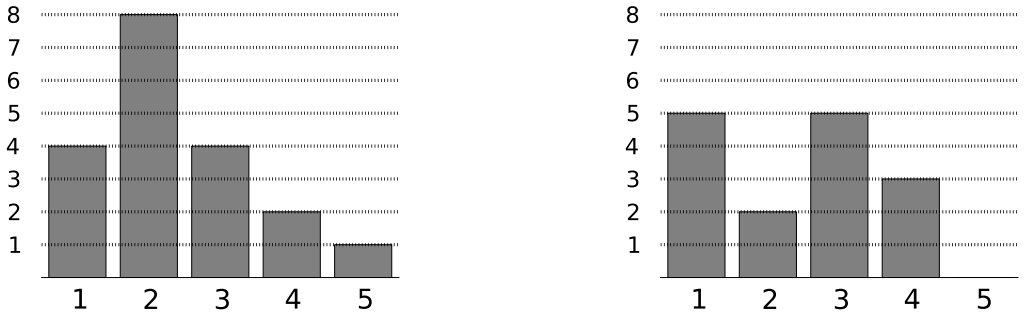
(a) Control Group **(b)** Experimental Group

Figure E.38: Histogram of the answers to the questionnaire item UN2 – *The way in which the information was organized and linked allowed me to understand it more easily.*



(a) Control Group **(b)** Experimental Group

Figure E.39: Histogram of the answers to the questionnaire item CO1 – *The information that was made available was often inconsistent.*



(a) Control Group **(b)** Experimental Group

Figure E.40: Histogram of the answers to the questionnaire item CO2 – *I don't have a good perception if the information that was available to me was consistent or not.*

Publications

Many of the materials in this thesis have appeared in the following publications.

Peer-reviewed Conference Papers[1]

[CA13] **F. F. Correia** and A. Aguiar, "Patterns of Flexible Modeling Tools", in *Proceedings of the 20th Conference on Pattern Languages of Programs* (PLoP), Allerton, Illinois, USA, 2013.

[MCY+11] P. Matsumoto, **F. F. Correia**, J. Yoder, E. Guerra, H. S. Ferreira, and A. Aguiar, "AOM Metadata Extension Points", in *Proceedings of the 18th Conference on Pattern Languages of Programs* (PLoP), Portland, Oregon, USA, 2011.

[CA11] **F. F. Correia** and A. Aguiar, "Patterns of Information Classification", in *Proceedings of the 18th Conference on Pattern Languages of Programs* (PLoP), Portland, Oregon, USA, 2011.

[FCAY11] H. S. Ferreira, **F. F. Correia**, A. Aguiar, and J. Yoder, "The Lazy Semantics Pattern on the context of Meta-Architectures", in *Proceedings of the 2nd Asian Conference on Pattern Languages of Programs* (PLoP), Tokyo, Japan, 2011.

[FCYA10] H. S. Ferreira, **F. F. Correia**, J. Yoder, and A. Aguiar, "Core Patterns of Object-Oriented Meta-Architectures", in *Proceedings of the 17th Conference on Pattern Languages of Programs* (PLoP), Reno, Nevada, USA, 2010.

[CFFA09b] **F. F. Correia**, H. S. Ferreira, N. Flores, and A. Aguiar, "Patterns for Consistent Software Documentation", in *Proceedings of the Pattern Languages of Programs* (PLoP), Chicago, Illinois, USA, 2009.

[FCW08] H. S. Ferreira, **F. F. Correia**, and L. Welicki, "Patterns for Data and Metadata Evolution in Adaptive Object-Models", in *Proceedings of the 15th Conference on Pattern Languages of Programs* (PLoP), Nashville, Tennessee, USA, 2008.

[1] The peer-review of papers accepted to the Pattern Language of Programs conference is based on a *shepherding* process and is followed by a writer's workshop. Both review techniques are a source of important feedback and actively support authors in improving their works.

Peer-reviewed Journal Papers

[FCAF10] H. S. Ferreira, **F. F. Correia**, A. Aguiar, and J. P. Faria, "Adaptive Object-Models: a Research Roadmap", *IARIA Journal*, 2010.

Peer-reviewed Workshop Papers

[CFFA12] **F. F. Correia**, N. Flores, H. S. Ferreira, and A. Aguiar, "Assessing Tools for Software Development — An overview of three user evaluations", *USER 2012 - User evaluation for Software Engineering Researchers Workshop*, Zürich, Switzerland, 2012. [not part of proceedings]

[FCA09] H. S. Ferreira, **F. F. Correia**, and A. Aguiar, "Design for an Adaptive Object-Model Framework: An Overview", in *4th Workshop on Models@run.time at MODELS 09*, 2009, pp. 71-80.

[SMA+09] A. R. Silva, D. Martinho, A. Aguiar, N. Flores, **F. F. Correia**, and H. S. Ferreira, "An Implementation Model for Agile Business Process Tools", *IWODE 2009 - International Workshop on Organizational Design and Engineering*, Lisbon, Portugal, 2009. [not part of proceedings]

[Cor08] **F. F. Correia**, "Extending and Integrating Wikis to Improve Software Documentation", *Wikis4SE - Wikis for Software Engineering Workshop @ WikiSum 2008*, Porto, Portugal, 2008. [not part of proceedings]

Peer-reviewed Posters

[Cor13] **F. F. Correia**, "Documenting Software Using Adaptive Software Artifacts", in *Proceedings of the 4th annual conference on Systems, programming, and applications: software for humanity* (SPLASH), Indianapolis, Indiana, USA, 2013.

[CFFA09a] **F. F. Correia**, H. S. Ferreira, N. Flores, and A. Aguiar, "Incremental Knowledge Acquisition in Software Development Using a Weakly-Typed Wiki", in *Proceedings of the 5th International Symposium on Wikis and Open Collaboration* (WikiSym), Orlando, Florida, USA, 2009.

Doctoral Symposiums

[Cor10] **F. F. Correia**, "Supporting the Evolution of Software Knowledge With Adaptive Software Artifacts", in *Proceedings of the ACM international conference companion on object oriented programming systems languages and applications companion* (OOPSLA), Reno, Nevada, USA, 2010, pp. 231-32.

[CA09] **F. F. Correia** and A. Aguiar, "Software Knowledge Capture and Acquisition: Tool Support for Agile Settings", in *Proceedings of the Fourth International Conference on Software Engineering Advances* (ICSEA), Porto, Portugal, 2009, pp. 542-547.

[CF08] **F. F. Correia** and H. S. Ferreira, "Trends on Adaptive Object Model Research", in *Proceedings of the Doctoral Symposium on Informatics Engineering 2008* (DSIE), Porto, Portugal, 2008.

References

[ABK+06] Steve Abrams, Bard Bloom, Paul Keyser, Doug Kimelman, Eric Nelson, Wendy Neuberger, Tova Roth, Ian Simmonds, Steven Tang, and John Vlissides, *Architectural thinking and modeling with the architects' workbench*, IBM Systems Journal **45** (2006), no. 3, 481–500. Cited on pp. 43 and 109.

[ACPT01] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella, *Design-code traceability recovery: selecting the basic linkage properties*, Science of Computer Programming **40** (2001), no. 2-3, 213 – 234. Cited on p. 32.

[AD05a] Ademar Aguiar and Gabriel David, *Patterns for documenting frameworks – part i*, Conference Proceedings Of The Second, Third And Fourth Nordic Conference On Pattern Languages Of Programs Vikingplop (Helsinki, Finland), September 2005. Cited on p. 65.

[AD05b] Ademar Aguiar and Gabriel David, *WikiWiki weaving heterogeneous software artifacts*, Proceedings of the 2005 international symposium on Wikis (San Diego, California, USA), ACM, 2005, pp. 67–74. Cited on pp. 11, 18, and 70.

[AD06a] Ademar Aguiar and Gabriel David, *Patterns for documenting frameworks – part II*, Proceedings of EuroPLoP 2006 (Irsee, Germany), July 2006. Cited on p. 65.

[AD06b] _____, *Patterns for documenting frameworks – part III*, Proceedings of the 2006 conference on Pattern languages of programs (Portland, Oregon, USA), October 2006. Cited on p. 65.

[AD06c] _____, *Patterns for documenting frameworks: customization*, Proceedings of the 2006 conference on Pattern languages of programs (Portland, Oregon, USA), ACM, 2006, pp. 1–10. Cited on p. 65.

[AD07] _____, *Patterns for documenting frameworks - process*, Proceedings of SugarLoafPLoP 2006 (Recife, Brazil), May 2007. Cited on pp. 14 and 65.

[AD11] _____, *Patterns for effectively documenting frameworks*, Transactions on Pattern Languages of Programming II (David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, James Noble, Ralph Johnson, Paris Avgeriou, Neil B. Harrison, and Uwe Zdun, eds.), vol. 6510, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 79–124. Cited on p. 14.

[ADP03] Ademar Aguiar, Gabriel David, and Manuel Padilha, *XSDoc: an extensible wiki-based infrastructure for framework documentation*, Jornadas de Ingeniería del Software y Bases de Datos (Alicante, Spain), October 2003. Cited on pp. 18 and 70.

[AEQ99] Jim Arlow, Wolfgang Emmerich, and John Quinn, *Literate modelling - capturing business knowledge with the UML*, The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, June 1998 (Mulhouse, France) (Jean Bézivin and Pierre-Alain Muller, eds.), vol. 1618, Springer, 1999, pp. 189–199. Cited on pp. 25, 26, and 113.

[AG05] Katja Andresen and Norbert Gronau, *An approach to increase adaptability in ERP systems*, Managing Modern Organizations with Information Technology: Proceedings of the 2005 Information RecourcesManagement Association International Conference (San Diego, California, USA), Idea Group Publishing, May 2005, pp. 883–885. Cited on pp. 39 and 41.

[Agu03] Ademar Aguiar, *A minimalist approach to framework documentation*, Ph.D. thesis, Faculdade de Engenharia da Universidade do Porto, September 2003. Cited on pp. 11, 16, 18, 24, 26, and 31.

[AL07] Sören Auer and Jens Lehmann, *What have innsbruck and leipzig in common? extracting semantics from wiki content*, 4th European Semantic Web Conference (ESWC 2007) (2007), 503–517. Cited on p. 18.

[Ale77] Christopher Alexander, *A pattern language: towns, buildings, construction*, Oxford University Press, New York, USA, 1977. Cited on pp. 59, 60, 61, and 187.

[Ale79] Christopher Alexander, *The timeless way of building*, Oxford University Press, New York, USA, 1979 (English). Cited on p. 59.

[Amb] Scott Ambler, *Agile/Lean Documentation: Strategies for Agile Software Development*, http://www.agilemodeling.com/essays/agileDocumentation.htm, [accessed on 2014/05/15]. Cited on p. 65.

[Amb02] ———, *Agile modeling: Effective practices for eXtreme programming and the unified process*, 1st ed., Wiley, April 2002. Cited on p. 65.

[AN04] Jim Arlow and Ila Neustadt, *Enterprise patterns and MDA: Building better software with archetype patterns and UML*, Addison-Wesley Professional, 2004. Cited on p. 26.

[And01] Hugh Anderson, *Formalization and 'literate' programming*, Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific, 2001, pp. 39–44. Cited on p. 25.

[ANS05] ANSI/NISO, *Z39.19. Guidelines for the construction, format, and management of monolingual controlled vocabularies*, 2005. Cited on p. 98.

[Arl06] Jim Arlow, *Increase the accessibility and comprehensibility of a visual model with literate modeling*, http://www.informit.com/articles/article.aspx?p=460398 [accessed on 2014/05/15], April 2006. Cited on p. 26.

[Arm00] Phillip G. Armour, *The five orders of ignorance*, Commun. ACM **43** (2000), no. 10, 17–20. Cited on p. 8.

[Ars00] Ali Arsanjani, *Rule object: A pattern language for adaptive and scalable business rule construction*, Proceedings of PLOP2000, 2000. Cited on p. 45.

[Art00] Eric Artzt, *Autoduck user's guide*, 2000. Cited on pp. 28 and 70.

[Atl] Atlassian, *Confluence — enterprise collaboration and wiki software*, http://www.atlassian.com/software/confluence/ [accessed on 2012/04/01]. Cited on p. 18.

[BA04] Kent Beck and Cynthia Andres, *Extreme programming explained: Embrace change (2nd edition)*, Addison-Wesley Professional, 2004. Cited on p. 9.

[BBvB+01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas, *Manifesto for agile software development*, http://agilemanifesto.org/ [accessed on 2008/06/24], 2001. Cited on pp. 2 and 9.

[BHS07] Frank Buschmann, Kevlin Henney, and Douglas C Schmidt, *Pattern-oriented software architecture: on patterns and pattern languages*, vol. 5, Wiley, Chichester, UK; Hoboken, New Jersey, USA, 2007 (English). Cited on pp. 60 and 187.

[BKM00] Greg Butler, Rudolf K. Keller, and Hafedh Mili, *A framework for framework documentation*, ACM Comput. Surv. **32** (2000), 15. Cited on p. 14.

[BM06] Joachim Bayer and Dirk Muthig, *A view-based approach for improving software documentation practices*, Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on, 2006, p. 10 pp. Cited on p. 70.

[BMNS11] Sabine Buckl, Florian Matthes, Christian Neubert, and Christian M. Schweda, *A lightweight approach to enterprise architecture modeling and documentation*, Information Systems Evolution (Will Aalst, John Mylopoulos, Norman M. Sadeh, Michael J. Shaw, Clemens Szyperski, Pnina Soffer, and Erik Proper, eds.), vol. 72, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 136–149. Cited on p. 21.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-oriented software architecture: a system of patterns*, vol. 1, Wiley, Chichester, New York, USA, 1996. Cited on p. 60.

[BPB10] Karin Breitman, Oscar Pastor, and Simone Barbosa, *Flexible narrative representations: Bridging the gap between formal models and informal representations*, ICSE 2010 Workshop on Flexible Modeling Tools (FlexiTools 2010) (Cape Town, South Africa), vol. 35, 2010, pp. 37–38. Cited on p. 113.

[Bri03] Lionel C. Briand, *Software documentation: how much is enough?*, Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on, 2003, pp. 13–15. Cited on p. 11.

[CA09] Filipe F. Correia and Ademar Aguiar, *Software knowledge capture and acquisition: Tool support for agile settings*, Fourth International Conference on Software Engineering Advances (Porto, Portugal), September 2009, pp. 542–547. Cited on p. 232.

[CA11] ———, *Patterns of information classification*, Proceedings of the 18th Conference on Pattern Languages of Programs (Portland, Oregon, USA), ACM, 2011. Cited on pp. 63 and 231.

[CA13] ———, *Patterns of flexible modeling tools*, Proceedings of the 20th Conference on Pattern Languages of Programs (Allerton, Illinois, USA), ACM, 2013. Cited on pp. 63 and 231.

[CB91] David Cordes and Markus Brown, *The literate-programming paradigm*, Computer **24** (1991), 52–61. Cited on p. 22.

[CDREP08] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio, *Automating co-evolution in model-driven engineering*, Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE, 2008, pp. 222–231. Cited on p. 106.

[CF08] Filipe F. Correia and Hugo S. Ferreira, *Trends on adaptive object model research*, Proceedings of the Doctoral Symposium on Informatics Engineering 2008 (Porto, Portugal), FEUP, January 2008. Cited on p. 232.

[CFFA09a] Filipe F. Correia, Hugo S. Ferreira, Nuno Flores, and Ademar Aguiar, *Incremental knowledge acquisition in software development using a weakly-typed wiki*, Proceedings of the 5th International Symposium on Wikis and Open Collaboration (Orlando, Florida, USA), ACM, October 2009. Cited on pp. 185 and 232.

[CFFA09b] ———, *Patterns for consistent software documentation*, Proceedings of the 16th Conference on Pattern Languages of Programs (Chicago, Illinois, USA), ACM, August 2009. Cited on pp. 63, 106, 110, 112, 114, and 231.

[CFFA12] Filipe F. Correia, Nuno Flores, Hugo S. Ferreira, and Ademar Aguiar, *Assessing tools for software development — an overview of three user evaluations*, USER 2012 Workshop (Zürich, Switzerland), 2012. Cited on p. 232.

[CHK+01] N. Chapin, J. E Hale, K. M Khan, J. F Ramil, and W. G Tan, *Types of software evolution and software maintenance*, Journal of Software Maintenance and Evolution Research and Practice **13** (2001), no. 1, 3–30. Cited on p. 35.

[CHRP03] Li-Te Cheng, Susanne Hupfer, Steven Ross, and John Patterson, *Jazzing up eclipse with collaborative tools*, Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange (Anaheim, California, USA), ACM, 2003, pp. 45–49. Cited on pp. 9 and 33.

[CJBV13] Jeffrey C. Carver, Natalia Juristo, Maria Teresa Baldassarre, and Sira Vegas, *Replications of software engineering experiments*, Empirical Software Engineering (2013), 1–10 (en). Cited on p. 159.

[CJMS10] Jeffrey Carver, Letizia Jaccheri, Sandro Morasca, and Forrest Shull, *A checklist for integrating student empirical studies with research and teaching goals*, Empirical Software Engineering **15** (2010), no. 1, 35–59. Cited on pp. 57, 154, 158, and 178.

[CKR10] Vanea Chiprianov, Yvon Kermarrec, and Siegfried Rouvrais, *Meta-tools for software language engineering: a flexible collaborative modeling language for efficient telecommunications service design*, ICSE 2010 Workshop on Flexible Modeling Tools (FlexiTools 2010) (Cape Town, South Africa), 2010. Cited on p. 111.

[Cor08] Filipe F. Correia, *Extending and integrating wikis to improve software documentation*, Wikis4SE WikiSym 2008 (Porto, Portugal), September 2008. Cited on p. 232.

[Cor10] _____, *Supporting the evolution of software knowledge with adaptive software artifacts*, Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion (Reno/Tahoe, Nevada, USA), SPLASH '10, ACM, 2010, pp. 231–232. Cited on pp. 103 and 232.

[Cor13] _____, *Documenting software using adaptive software artifacts*, Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, SPLASH '13, ACM, 2013. Cited on pp. 103 and 232.

[CS96] Bart Childs and Johannes Sametinger, *Literate programming and documentation reuse*, Software Reuse, 1996., Proceedings Fourth International Conference on, 1996, pp. 205–214. Cited on p. 76.

[CSGW11] Hyun Cho, Y. Sun, J. Gray, and Jules White, *Key challenges for modeling language creation by demonstration*, ICSE 2011 Workshop on Flexible Modeling Tools (FlexiTools 2011) (Waikiki, Hawaii, USA), 2011. Cited on p. 108.

[Cuna] Ward Cunningham, *c2.com — wiki*, http://c2.com/cgi/wiki [accessed on 2007/12/01]. Cited on p. 15.

[Cunb] _____, *c2.com — wiki design principles*, http://c2.com/cgi/wiki?WikiDesignPrinciples [accessed on 2007/12/01]. Cited on pp. 15 and 185.

[Cun93] _____, *The WyCash portfolio management system*, SIGPLAN OOPS Mess. **4** (1993), no. 2, 29–30. Cited on p. 36.

[Cun06] _____, *Design principles of wiki: how can so little do so much?*, http://c2.com/doc/wikisym/WikiSym2006.pdf [accessed on 2014/05/15], 2006. Cited on p. 15.

[Dav05] Thomas H. Davenport, *Thinking for a living: How to get better performances and results from knowledge workers*, Harvard Business Press, September 2005. Cited on pp. 8 and 186.

[DGB07] Sergiu Dumitriu, Marta Gîrdea, and Sabin C. Buraga, *From information wiki to knowledge wiki via semantic web technologies*, Innovations and Advanced Techniques in Computer and Information Sciences and Engineering (Tarek Sobh, ed.), Springer Netherlands, January 2007, pp. 443–448. Cited on p. 21.

[DGLP08] Marco D'Ambros, Harald Gall, Michele Lanza, and Martin Pinzger, *Analysing software repositories to understand software evolution*, Software Evolution, Springer, 2008, pp. 37–67. Cited on p. 38.

[DOS10] Michael Desmond, Harold Ossher, and Ian Simmonds, *Towards smart office tools*, SPLASH 2010 Workshop on Flexible Modeling Tools (FlexiTools 2010@SPLASH) (Reno, Nevada, USA), 2010. Cited on p. 108.

[DRR+05] Björn Decker, Eric Ras, Jörg Rech, Bertin Klein, and Christian Hoecht, *Self-organized reuse of software engineering knowledge supported by semantic wikis*, Proceedings of the Workshop on Semantic Web Enabled Software Engineering (SWESE), November 2005. Cited on p. 76.

[DV99] Angela M. Dean and Daniel Voss, *Design and analysis of experiments*, 1st ed. 1999. corr. 2nd printing ed., Springer, 1999. Cited on pp. 57, 154, and 159.

[Edga] Edgewall Software, *The Trac Project — Integrated SCM & Project Management*, `http://trac.edgewall.org/` [accessed on 2009/12/01]. Cited on pp. 3, 18, 19, 77, and 132.

[Edgb] _____, *Trac Component Architecture*, `http://trac.edgewall.org/wiki/TracDev/ComponentArchitecture` [accessed on 2012/04/01]. Cited on p. 134.

[Edgc] _____, *Trac Extension points*, `http://trac.edgewall.org/wiki/TracDev/PluginDevelopment/ExtensionPoints` [accessed on 2012/04/01]. Cited on p. 134.

[Eva03] Eric Evans, *Domain-Driven design: Tacking complexity in the heart of software*, 1 ed., Addison Wesley, 2003. Cited on p. 40.

[FB99] Martin Fowler and Kent Beck, *Refactoring: improving the design of existing code*, Addison-Wesley Professional, 1999. Cited on p. 37.

[FCA09] Hugo S. Ferreira, Filipe F. Correia, and Ademar Aguiar, *Design for an adaptive object-model framework: An overview*, 4th Workshop on Modelsrun.time at MODELS 09, October 2009, pp. 71–80. Cited on pp. 106, 136, 142, and 232.

[FCAF10] Hugo S. Ferreira, Filipe F. Correia, Ademar Aguiar, and João Pascoal Faria, *Adaptive object-models: a research roadmap*, IARIA Journal (2010). Cited on pp. 42 and 232.

[FCAY11] Hugo S. Ferreira, Filipe F. Correia, Ademar Aguiar, and Joseph Yoder, *The lazy semantics pattern on the context of meta-architectures*, 2011. Cited on pp. 115 and 231.

[FCW08] Hugo S. Ferreira, Filipe F. Correia, and Leon Welicki, *Patterns for data and metadata evolution in adaptive object-models*, Proceedings of the 15th Conference on Pattern Languages of Programs (Nashville, Tennessee, USA), ACM, October 2008. Cited on pp. 38, 103, 106, 114, 116, and 231.

[FCYA10] Hugo S. Ferreira, Filipe F. Correia, Joseph Yoder, and Ademar Aguiar, *Core patterns of object-oriented meta-architectures*, Proceedings of the 17th Conference on Pattern Languages of Programs (Reno, Nevada, USA), ACM, 2010. Cited on pp. 103, 104, 114, 115, and 231.

[Fer10] Hugo S. Ferreira, *Adaptive object-modeling: Patterns, tools and applications*, PhD in computer science, University of Porto, Faculty of Engineering, Rua Dr. Roberto Frias, s/n, 4200-4650 Porto, Portugal, 2010. Cited on pp. 44 and 114.

[fit] *Fitnesse*, http://fitnesse.org/ [accessed on 2011/01/02]. Cited on pp. 18 and 19.

[Fow97] Martin Fowler, *Analysis patterns: reusable objects models*, Addison-Wesley Longman Publishing Co., Inc, Boston, Massachusetts, USA, 1997. Cited on p. 45.

[Fow06] Martin Fowler, *Writing software patterns*, http://www.martinfowler.com/articles/writingPatterns.html, [accessed on 2008/09/13], 2006. Cited on p. 61.

[Fri95] Lisa Friendly, *The design of distributed hyperlinked programming documentation*, IWHD'95 (Montpellier, France), 1995. Cited on pp. 28, 70, and 75.

[Fur10] Jonathan Furner, *Folksonomies*, Encyclopedia of Library and Information Sciences (Marcia J. Bates and Mary Niles Maack, eds.), CRC Press, Boca Raton, Florida, USA, 3rd ed., 2010, pp. 1858–1866. Cited on p. 95.

[FY99] Brian Foote and Joseph Yoder, *Big ball of mud*, Pattern Languages of Program Design 4, Addison-Wesley, 1999, pp. 653–692. Cited on p. 36.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom, *Architectural mismatch: Why reuse is so hard*, Software, IEEE **12** (1995), no. 6, 17–26. Cited on p. 12.

[GBeA07] Miguel Goulão and Fernando Brito e Abreu, *Modeling the experimental software engineering process*, Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the, 2007, pp. 77–90. Cited on p. 160.

[GG08] Michael W. Godfrey and Daniel M. German, *The past, present, and future of software evolution*, Frontiers of Software Maintenance, 2008. FoSM 2008., 2008, pp. 129–138. Cited on pp. 35 and 38.

[GGLS11] Gregor Gabrysiak, Holger Giese, Alexander Lueders, and Andreas Seibel, *How can metamodels be used flexibly?*, ICSE 2011 Workshop on Flexible Modeling Tools (FlexiTools 2011) (Waikiki, Hawaii, USA), 2011. Cited on pp. 103, 104, 105, 107, and 112.

[GGS10] Gregor Gabrysiak, Holger Giese, and Andreas Seibel, *Using ontologies for flexibly specifying multi-user processes*, ICSE 2010 Workshop on Flexible Modeling Tools (FlexiTools 2010) (Cape Town, South Africa), 2010. Cited on p. 105.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley Professional, January 1995. Cited on pp. 60, 61, and 105.

[GJT09] Raghu Garud, Sanjay Jain, and Philipp Tuertscher, *Incomplete by design and designing for incompleteness*, Design Requirements Engineering: A Ten-Year Perspective, Springer, 2009, pp. 137–156. Cited on p. 35.

[GM12] Gary Goertz and James Mahoney, *A tale of two cultures: qualitative and quantitative research in the social sciences*, Princeton University Press, Princeton, N.J., 2012 (English). Cited on p. 57.

[GR13] Olivier Gendreau and Pierre N. Robillard, *Knowledge acquisition activity in software development*, Advances in Information Systems and Technologies (Álvaro Rocha, Ana Maria Correia, Tom Wilson, and Karl A. Stroetmann, eds.), Advances in Intelligent Systems and Computing, no. 206, Springer Berlin Heidelberg, January 2013, pp. 1–10. Cited on p. 8.

[GRS12] Chiara Ghidini, Marco Rospocher, and Luciano Serafini, *Conceptual modeling in wikis: a reference architecture and a tool*, eKNOW 2012, The Fourth International Conference on Information, Process, and Knowledge Management, 2012, pp. 128–135. Cited on p. 21.

[Gru93] Thomas R. Gruber, *A translation approach to portable ontology specifications*, Knowledge Acquisition **5** (1993), no. 2, 199–220. Cited on p. 92.

[Ham94] John Hamer, *Literate programming: a software engineering perspective*, Software Education Conference, 1994. Proceedings., 1994, pp. 282–288. Cited on pp. 22, 24, and 26.

[HHT01] Jochen Hartmann, Shihong Huang, and Scott Tilley, *Documenting software systems with views II: an integrated approach based on XML*, Proceedings of the 19th annual international conference on Computer documentation (Sante Fe, New Mexico, USA), ACM, 2001, pp. 237–246. Cited on p. 14.

[HNA+10] Jonas Helming, Nitesh Narayan, Holger Arndt, Maximilian Koegel, and Walid Maalej, *From informal project management artifacts to formal system models*, ICSE 2010 Workshop on Flexible Modeling Tools (FlexiTools 2010) (Cape Town, South Africa), 2010. Cited on p. 110.

[HS98] Nick Hatzigeorgiu and Apostolos Syropoulos, *Literate programming and the "spaniel" method*, SIGPLAN Not. **33** (1998), 52–56. Cited on p. 26.

[HW99] Myles Hollander and Douglas A. Wolfe, *Nonparametric statistical methods*, 2nd ed., Wiley-Interscience, January 1999. Cited on p. 171.

[IVZ08] Angelo Di Iorio, Fabio Vitali, and Stefano Zacchiroli, *Wiki content templating*, Proceeding of the 17th international conference on World Wide Web (Beijing, China), ACM, 2008, pp. 615–624. Cited on pp. 17 and 18.

[IVZ09] Angelo Di Iorio, Fabio Vitali, and S. Zacchiroli, *Web semantics via wiki templating*, Handbook of research on Web 2.0, 3.0 and x.0: technologies, business and social applications, Advances in E-Business Research, San Murugesan Ed., November 2009. Cited on p. 18.

[Job93] Chris P. Jobling, *The IEEE grumman f14 benchmark problem - an example of literate modelling in ACSL using noweb*, 1993. Cited on p. 113.

[JW97] Ralph Johnson and Bobby Woolf, *The type object pattern*, 1997. Cited on pp. 45 and 136.

[Kac12] Michal Kacprzyk, *Software knowledge management using wikis : a plugin for weakly typed pages*, Ph.D. thesis, Faculdade de Engenharia da Universidade do Porto, Porto, Portugal, February 2012. Cited on p. 185.

[Kan] Noah Kantrowitz, *TracHacks — IncludeMacro Plugin*, `http://trac-hacks.org/wiki/IncludeMacro` [accessed on 2012/04/01]. Cited on p. 19.

[KB03] Michael A. Katz and Michael D. Byrne, *Effects of scent and breadth on use of site-specific search on e-commerce web sites*, ACM Transactions on Computer-Human Interaction **10** (2003), no. 3, 198–220 (en). Cited on p. 52.

[KC02] Andreas Kacofegitis and Neville Churcher, *Theme-based literate programming*, Software Engineering Conference, 2002. Ninth Asia-Pacific, 2002, pp. 549–557. Cited on pp. 22, 24, 27, and 28.

[KCM07] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic, *A survey and taxonomy of approaches for mining software repositories in the context of software evolution*, J. Softw. Maint. Evol. **19** (2007), no. 2, 77–131. Cited on p. 38.

[KH10] Doug Kimelman and Ken Hirschman, *Discussion and MindManager-Based structuring of workshop summary*, `http://www.ics.uci.edu/~nlopezgi/flexitools/presentations/SPLASH2010WorkshoponFlexibleModelingTools-v6.swf`, [accessed on 2014/05/15], 2010. Cited on p. 103.

[KH11] _____, *A spectrum of flexibility - lowering barriers to modeling tool adoption*, ICSE 2011 Workshop on Flexible Modeling Tools (FlexiTools 2011) (Waikiki, Hawaii, USA), 2011. Cited on pp. 43 and 103.

[KL02] Donald E. Knuth and Silvio Levy, *The CWEB system of structured documentation — version 3.64*, 2002. Cited on p. 22.

[Kna96] Markus Knasmueller, *Reverse literate programming*, Proceedings of the Software Quality Conference (Dundee, Scotland, UK), Johannes Kepler Universitat Linz, 1996. Cited on pp. 24 and 27.

[Knu83] Donald E. Knuth, *The WEB system of structured documentation*, 1983. Cited on pp. 22 and 26.

[Knu84] _____, *Literate programming*, Comput. J. **27** (1984), no. 2, 97–111. Cited on pp. 22 and 70.

[Koc05] Stefan Koch, *Evolution of open source software systems — a large-scale investigation*, Proceedings of the 1st International Conference on Open Source Systems (Genova, Italy), July 2005, pp. 148–153. Cited on p. 38.

[KOvdHS10] Doug Kimelman, Harold Ossher, André van der Hoek, and Margaret-Anne Storey, *SPLASH 2010 workshop on flexible modeling tools*, Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion (Reno, Nevada, USA), SPLASH '10, ACM, 2010, pp. 283–284. Cited on pp. 43 and 102.

[KP10] Christian Kohls and Stefanie Panke, *Is that true...?: thoughts on the epistemology of patterns*, Proceedings of the 16th Conference on Pattern Languages of Programs (Chicago, Illinois, USA), PLoP '09, ACM, 2010, pp. 9:1–9:14. Cited on pp. 60 and 187.

[Kuh11] Marco Kuhrmann, *User assistance during domain-specific language design*, ICSE 2011 Workshop on Flexible Modeling Tools (FlexiTools 2011) (Waikiki, Hawaii, USA), 2011. Cited on p. 108.

[KVV06] Markus Krötzsch, Denny Vr, and Max Völkel, *Semantic MediaWiki*, Proceedings of the 5TH International Semantic Web Conference (ISWC06), vol. 4273, 2006, pp. 935–942. Cited on p. 20.

[Lan] Jean-Philippe Lang, *Redmine*, `http://www.redmine.org/`, [accessed on 2009/12/01]. Cited on p. 77.

[Lan03] Frederick Wilfrid Lancaster, *Indexing and abstracting in theory and practice*, 3rd ed., University of Illinois Graduate School of Library and Information Science, 2003. Cited on pp. 80 and 83.

[LB85] Manny M. Lehman and Les A. Belady (eds.), *Program evolution: processes of software change*, Academic Press Professional, Inc., 1985. Cited on p. 35.

[LC01] Bo Leuf and Ward Cunningham, *The wiki way: quick collaboration on the web*, Addison-Wesley Professional, 2001. Cited on p. 15.

[Lik32] Rensis. Likert, *A technique for the measurement of attitudes.*, Archives of psychology (1932). Cited on p. 156.

[LP88] T. F. Lunney and R. H. Perrott, *Syntax-directed editing*, Software Engineering Journal **3** (1988), no. 2, 37–46. Cited on p. 32.

[LPPS08] Miguel Luaces, Jose Paramá, Oscar Pedreira, and Diego Seco, *An ontology-based index to retrieve documents with geographic information*, Scientific and Statistical Database Management, 2008, pp. 384—400. Cited on p. 93.

[MBZR03] Tom Mens, Jim Buckley, Matthias Zenger, and Awais Rashid, *Towards a taxonomy of software evolution*, Proceedings of Workshop on Unanticipated Software Evolution (Warsaw, Poland), 2003. Cited on pp. 35, 37, and 39.

[MCY+11] Patricia. Matsumoto, Filipe F. Correia, Joseph Yoder, Eduardo Guerra, Hugo S. Ferreira, and A. Aguiar, *AOM metadata extension points*, Proceedings of the 18th Conference on Pattern Languages of Programs (Portland, Oregon, USA), 2011. Cited on p. 231.

[Men08] Tom Mens, *Introduction and roadmap: History and challenges of software evolution*, Software Evolution, Springer, 2008, pp. 1–11. Cited on pp. 32 and 35.

[Mer] Matteo Merli, *Trac wiki editor for eclipse*, http://trac-hacks.org/wiki/EclipseTracPlugin, [accessed on 2010/08/03]. Cited on p. 38.

[MNS11] Florian Matthes, Christian Neubert, and Alexander Steinhoff, *Hybrid wikis: Empowering users to collaboratively structure information*, 6th International Conference on Software and Data Technologies (ICSOFT), Seville, July 2011. Cited on p. 21.

[Mul96] Multiple Authors, *KM forum discussion archives - knowledge vs information*, http://www.km-forum.org/t000008.htm, [accessed on 2014/05/15], 1996. Cited on p. 8.

[MWD+05] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri, *Challenges in software evolution*, Principles of Software Evolution, Eighth International Workshop on, 2005, pp. 13–22. Cited on p. 35.

[Myl] Mylyn Community, *Eclipse mylyn open source project*, http://www.eclipse.org/mylyn/, [accessed on 2010/08/03]. Cited on p. 38.

[Mäo5] Eetu Mäkelä, *Survey of semantic search research*, Proceedings of the seminar on knowledge management on the semantic web, Department of Computer Science, University of Helsinki, Helsinki, 2005. Cited on p. 152.

[NHKN10] Michael Nagel, Jonas Helming, Maximilian Koegel, and Helmut Naughton, *Audio recording in software engineering*, ICSE 2010 Workshop on Flexible Modeling Tools (FlexiTools 2010) (Cape Town, South Africa), 2010. Cited on p. 113.

[Nie93] Jakob Nielsen, *Usability engineering*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. Cited on p. 159.

[Nøoo] Kurt Nørmark, *Requirements for an elucidative programming environment*, Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on, 2000, pp. 119–128. Cited on p. 29.

[OBA+09] Harold Ossher, Rachel Bellamy, David Amid, Ateret Anaby-Tavor, Mathew Callery, Michael Desmond, Jaqueline de Vries, Amit Fisher, Thomas Frauenhofer, Sophia Krasikov, Ian Simmonds, and Calvin Swart, *Business insight toolkit: Flexible pre-requirements modeling*, 31st International Conference on Software Engineering - Companion Volume, 2009. ICSE-Companion 2009, IEEE, May 2009, pp. 423–424. Cited on p. 43.

[OBS+10] Harold Ossher, Rachel Bellamy, Ian Simmonds, David Amid, Ateret Anaby-Tavor, Mathew Callery, Michael Desmond, Jaqueline de Vries, Amit Fisher, and Sophia Krasikov, *Flexible modeling tools for pre-requirements analysis: conceptual architecture and research challenges*, Proceedings of the ACM international conference on Object oriented programming systems languages and applications, 2010, pp. 848–864. Cited on p. 44.

[OMG] OMG, *MOF — MetaObject Facility*, http://www.omg.org/mof/ [accessed on 2007/12/01]. Cited on p. 42.

[OvdHS09] Harold Ossher, Andre van der Hoek, and Margaret-Anne Storey, *Flexible modeling tools*, Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research (Toronto, Ontario, Canada), CASCON '09, ACM, 2009, pp. 350–352. Cited on p. 43.

[OvdHS+10a] Harold Ossher, André van der Hoek, Margaret-Anne Storey, John Grundy, and Rachel Bellamy, *Flexible modeling tools (FlexiTools2010)*, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (Cape Town, South Africa), ICSE '10, ACM, 2010, pp. 441–442. Cited on p. 43.

[OvdHS+10b] Harold Ossher, André van der Hoek, Margaret-Anne Storey, John Grundy, and Rachel Bellamy, *Flexible modeling tools (FlexiTools2010)*, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (Cape Town, South Africa), ICSE '10, ACM, 2010, pp. 441–442. Cited on p. 102.

[OvdHS+11] Harold Ossher, Andre van der Hoek, Margaret-Anne Storey, John Grundy, Rachel Bellamy, and Marian Petre, *Workshop on flexible modeling tools: (FlexiTools 2011)*, 2011 33rd International Conference on Software Engineering (ICSE), IEEE, May 2011, pp. 1192–1193. Cited on pp. 43 and 102.

[Par94] David Lorge Parnas, *Software aging*, Proceedings of the 16th international conference on Software engineering (Sorrento, Italy), IEEE Computer Society Press, 1994, pp. 279–287. Cited on p. 35.

[PKB04] Vreda Pieterse, Derrick G. Kourie, and Andrew Boake, *A case for contemporary literate programming*, Proceedings of the 2004 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries (Stellenbosch, Western Cape, South Africa), South African Institute for Computer Scientists and Information Technologists, 2004, pp. 2–9. Cited on pp. 22 and 26.

[Rö3] Andreas Rüping, *Agile documentation: A pattern guide to producing lightweight documents for software projects*, Wiley, September 2003. Cited on pp. 16, 65, 70, 73, 75, and 77.

[Raj97] Vaclav Rajlich, *A model for change propagation based on graph rewriting*, ICSM '97: Proceedings of the International Conference on Software Maintenance (Washington, DC, USA), IEEE Computer Society, 1997, pp. 84–91. Cited on p. 37.

[Ram94] Norman Ramsey, *Literate programming simplified*, Software, IEEE **11** (1994), 97–105. Cited on pp. 22 and 25.

[RAMF04] Heather Richter, Gregory Abowd, Chris Miller, and Harry Funk, *Tagging knowledge acquisition sessions to facilitate knowledge traceability*, International Journal of Software Engineering and Knowledge Engineering **14** (2004), no. 1, 3–19. Cited on p. 8.

[RBJ97] Don Roberts, John Brant, and Ralph Johnson, *A refactoring tool for smalltalk*, Theory and Practice of Object Systems **3** (1997), 253–263. Cited on p. 32.

[REM+09] Dirk Riehle, John Ellenberger, Tamir Menahem, Boris Mikhailovski, Yuri Natchetoi, Barak Naveh, and Thomas Odenwald, *Open collaboration within corporations using software forges*, IEEE Softw. **26** (2009), no. 2, 52–58. Cited on pp. 30 and 32.

[RFBO01] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe, *The architecture of a UML virtual machine*, OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (Tampa Bay, Florida, USA), ACM, 2001, pp. 327–341. Cited on p. 41.

[RG04] Vaclav Rajlich and Prashant Gosavi, *Incremental change in object-oriented programming*, IEEE Softw. **21** (2004), no. 4, 62–69. Cited on p. 37.

[RL04] Awais Rashid and Nicholas Leidenfrost, *Supporting flexible object database evolution with aspects*, 3rd Int. Conf. on Generative Programming and Component Engineering (GPCE 2004), volume 3286 of LNCS, Springer, 2004, pp. 75–94. Cited on p. 38.

[Rob99] Pierre N. Robillard, *The role of knowledge in software development*, Communications of the ACM **42** (1999), no. 1, 87–92. Cited on p. 8.

[Rub] Ruby on Rails Community, *Migrations*, `http://guides.rubyonrails.org/migrations.html` [accessed on 2010/07/30]. Cited on p. 38.

[Sam94] Johannes Sametinger, *Object-oriented documentation*, SIGDOC Asterisk J. Comput. Doc. **18** (1994), no. 1, 3–14. Cited on p. 76.

[SB10] Ugo Braga Sangiorgi and SD Barbosa, *SKETCH: modeling using freehand drawing in eclipse graphical editors*, ICSE 2010 Workshop on Flexible Modeling Tools (FlexiTools 2010) (Cape Town, South Africa), 2010. Cited on p. 109.

[SBC+02] Forrest Shull, Victor Basili, Jeffrey Carver, José Carlos Maldonado, Guilherme Horta Travassos, Manoel Mendonça, and Sandra Fabbri, *Replicating software engineering experiments: addressing the tacit knowledge problem*, Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium n, 2002, pp. 7–16. Cited on p. 160.

[SBD+08] Sebastian Schaffert, Francois Bry, Peter Dolog, Julia Eder, Szaby Gruenwald, Jana Herwig, Jozef Holy, Peter Axel Nielsen, and Pavel Smrz, *KiWi vision: Collaborative knowledge management, powered by the semantic web*, Tech. report, Salzsburg Research Forschungsgesellschaft, 2008. Cited on p. 20.

[SC93] Stephen Shum and Curtis Cook, *AOPS: an abstraction-oriented programming system for literate programming*, Software Engineering Journal **8** (1993), 113–120. Cited on p. 25.

[SCBR06] Margaret-Anne Storey, Li-Te Cheng, Ian Bull, and Peter Rigby, *Shared waypoints and social tagging to support collaboration in software development*, Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work (Banff, Alberta, Canada), ACM, 2006, pp. 195–198. Cited on p. 32.

[SCDLG12] Jesús Sánchez-Cuadrado, Juan De Lara, and Esther Guerra, *Bottom-up meta-modelling: An interactive approach*, Model Driven Engineering Languages and Systems, Springer, 2012, pp. 3–19. Cited on p. 107.

[Sch06] Douglas C. Schmidt, *Guest editor's introduction: Model-Driven engineering*, Computer **39** (2006), no. 2, 25–31. Cited on p. 41.

[SHH+05] Dag I. K. Sjoeberg, Jo E. Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, Nils-Kristian Liborg, and Anette Rekdal, *A survey of controlled experiments in software engineering*, IEEE Transactions on Software Engineering **31** (2005), no. 9, 733–753. Cited on p. 159.

[SMA+09] António Rito Silva, David Martinho, Ademar Aguiar, Nuno Flores, Filipe F. Correia, and Hugo S. Ferreira, *An implementation model for agile business process tools*, 2009. Cited on p. 232.

[Smi01] Mathew Smith, *Towards modern literate programming*, 2001. Cited on pp. 26 and 31.

[Sou05] Alexandre Sousa, *dotNoweb User's Guide*, Tech. report, 2005. Cited on p. 70.

[Spo02] Joel Spolsky, *The law of leaky abstractions*, `http://www.joelonsoftware.com/articles/LeakyAbstractions.html` [accessed on 2014/05/15], November 2002. Cited on p. 24.

[Sta10] Michael Stal, *Implicit versus explicit*, http://stal.blogspot.pt/2010/03/implicit-versus-explicit.html [accessed on 2012/12/01], March 2010. Cited on p. 12.

[STvDC10] Margaret-Anne Storey, Christoph Treude, Arie van Deursen, and Li-Te Cheng, *The impact of social media on software engineering practices and tools*, Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (Santa Fe, New Mexico, USA), FoSER '10, ACM, 2010, pp. 359–364. Cited on p. 9.

[SV06] Thomas Stahl and Markus Voelter, *Model-Driven software development: Technology, engineering, management*, 1 ed., Wiley, May 2006. Cited on pp. 13, 41, and 99.

[SvG05] Margaret-Anne D. Storey, Davor Čubranić, and Daniel M. German, *On the use of visualization to support awareness of human activities in software development: a survey and a framework*, Proceedings of the 2005 ACM symposium on Software visualization (St. Louis, Missouri, USA), ACM, 2005, pp. 193–202. Cited on p. 32.

[Swa76] E. Burton Swanson, *The dimensions of maintenance*, Proceedings of the 2nd international conference on Software engineering (San Francisco, California, USA), IEEE Computer Society Press, 1976, pp. 492–497. Cited on p. 35.

[SZ01] George Spanoudakis and Andrea Zisman, *Inconsistency management in software engineering: Survey and open research issues*, in Handbook of Software Engineering and Knowledge Engineering, World Scientific, 2001, pp. 329–380. Cited on p. 37.

[TAAK04] Jaime Teevan, Christine Alvarado, Mark S. Ackerman, and David R. Karger, *The perfect search engine is not enough: a study of orienteering behavior in directed search*, ACM Press, 2004, pp. 415–422 (en). Cited on p. 52.

[Ten08] Joseph T. Tennis, *Organization of information and resources — lecture materials on pre- and post-coordinate indexing*, 2008. Cited on p. 80.

[Thi86] Harold Thimbleby, *Experiences of 'literate programming' using cweb (a variant of knuth's WEB)*, The Computer Journal **29** (1986), 201–211. Cited on pp. 24, 25, and 26.

[Tho06] Dave Thomas, *Programming with models - modeling with code — the role of models in software development*, Journal of Object Technology **5** (2006), 15–19. Cited on p. 26.

[TPK07] Juha-Pekka Tolvanen, Risto Pohjonen, and Steven Kelly, *Advanced tooling for domain-specific modeling: MetaEdit+*, Sprinkle, J., Gray, J., Rossi, M., Tolvanen, JP (eds.) The 7th OOPSLA Workshop on Domain-Specific Modeling, Finland, 2007. Cited on p. 104.

[TPT09] Susanna Teppola, Päivi Parviainen, and Juha Takalo, *Challenges in deployment of model driven development*, Fourth International Conference on Software Engineering Advances, 2009. ICSEA '09, IEEE, September 2009, pp. 15–20. Cited on p. 13.

[USE] USER 2012 organizers and expert panel, *Handout for user workshop (user evaluation for software engineering researchers), ICSE 2012*. Cited on p. 57.

[vAK92] Eric van Ammers and Mark Kramer, *VAMP: A tool for literate programming independent of programming language and formatter*, CompEuro '92 . 'Computer Systems and Software Engineering',Proceedings., 1992, pp. 371–376. Cited on pp. 22, 25, and 26.

[Ves03] Thomas Vestdam, *Elucidative programming in open integrated development environments for java*, Proceedings of the 2nd international conference on Principles and practice of programming in Java (Kilkenny City, Ireland), Computer Science Press, Inc., 2003, pp. 49–54. Cited on p. 29.

[vGBS01] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg, *On the notion of variability in software product lines*, Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on (Amsterdam, Netherlands), 2001, pp. 45–54. Cited on p. 41.

[vH97] Dimitri van Heesch, *Doxygen home page*, http://doxygen.org/ [accessed on 2012/03/20], 1997. Cited on p. 28.

[VJ10] Bernhard Volz and Stefan Jablonski, *OMME - a flexible modeling environment*, SPLASH 2010 Workshop on Flexible Modeling Tools (FlexiTools 2010@SPLASH) (Reno, Nevada, USA), 2010. Cited on p. 111.

[VN02] Thomas Vestdam and Kurt Nørmark, *Aspects of internal program documentation-an elucidative perspective*, Program Comprehension, 2002. Proceedings. 10th International Workshop on, 2002, pp. 43–52. Cited on pp. 29 and 70.

[VN04] _____, *Maintaining program understanding: issues, tools, and future directions*, Nordic J. of Computing **11** (2004), 303–320. Cited on p. 14.

[VN05] _____, *Toward documentation of program evolution*, Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on, 2005, pp. 505–514. Cited on p. 30.

[VZJ11] Bernhard Volz, Michael Zeising, and Stefan Jablonski, *The open meta modeling environment*, ICSE 2011 Workshop on Flexible Modeling Tools (FlexiTools 2011) (Waikiki, Hawaii, USA), 2011. Cited on p. 111.

[W3C08] W3C, *Resource description framework*, http://www.w3.org/RDF/, [accessed on 2014/05/15], March 2008. Cited on p. 20.

[Wac07] Guido Wachsmuth, *Metamodel adaptation and model co-adaptation*, ECOOP 2007–Object-Oriented Programming, Springer, 2007, pp. 600–624. Cited on p. 106.

[WE00] Han-Chieh Wei and Ramez Elmasri, *Schema versioning and database conversion techniques for bi-temporal databases*, Annals of Mathematics and Artificial Intelligence **30** (2000), no. 1-4, 23–52. Cited on p. 38.

[Wel94] Hans Wellisch, *Indexing*, Encyclopedia of Library History, Garland, New York, USA, 1994, pp. 268–270. Cited on p. 77.

[Wir08] Rebecca Wirfs-Brock, *Designing then and now*, IEEE Software **25** (2008), no. 6, 29–31. Cited on p. 43.

[Wyk89] Christopher J. Van Wyk, *Literate programming: Weaving a language independent WEB*, Commun. ACM **32** (1989), 1051–1055. Cited on p. 22.

[Wyk90] _____, *Literate programming: An assessment*, Commun. ACM **33** (1990), no. 3, 361–365. Cited on p. 23.

[WYW07] León Welicki, Joseph W. Yoder, and Rebecca Wirfs-Brock, *Rendering patterns for adaptive Object-Models*, 14th Pattern Language of Programs Conference (PLoP 2007) (Monticello, Illinois, USA), 2007. Cited on pp. 44 and 142.

[WYWJ07] León Welicki, Joseph W. Yoder, Rebecca Wirfs-Brock, and Ralph E. Johnson, *Towards a pattern language for adaptive object models*, Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion (Montréal, Québec, Canada), ACM, 2007, pp. 787–788. Cited on pp. 44 and 114.

[XCY07] WenPeng Xiao, ChangYan Chi, and Min Yang, *On-line collaborative software development via wiki*, Proceedings of the 2007 international symposium on Wikis (Montréal, Québec, Canada), ACM, 2007, pp. 177–183. Cited on p. 19.

[YBJ01] Joseph W. Yoder, Federico Balaguer, and Ralph Johnson, *Architecture and design of adaptive object-models*, ACM SIG-PLAN Notices **36** (2001), no. 12, 50–60. Cited on pp. 39, 45, 48, and 136.

[YFRT98] Joseph W. Yoder, Brian Foote, Dirk Riehle, and Michel Tilman, *Metadata and active object-models*, ACM, 1998. Cited on pp. 45 and 114.

[ZVB11] Li Zhu, Ivan Vaghi, and Barbara Rita Barricelli, *A meta-reflective wiki for collaborative design*, Proceedings of the 7th International Symposium on Wikis and Open Collaboration (Mountain View, California, USA), WikiSym '11, ACM, October 2011, pp. 53–62. Cited on p. 21.

Index